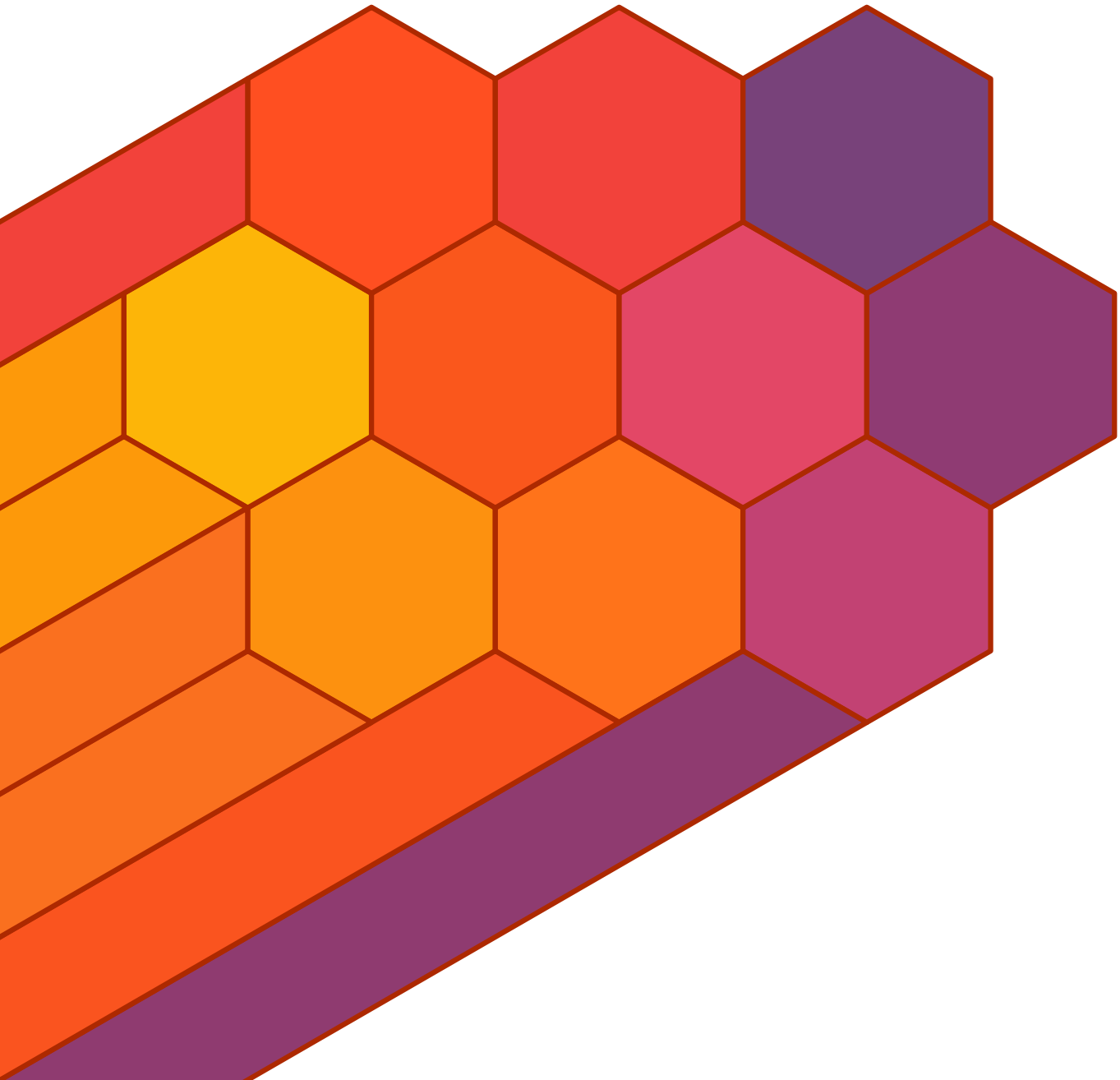
ucsd-psystem-xc
UCSD p-System Cross Compiler

Reference Manual

Peter Miller
<*pmiller@opensource.org.au*>



.

This document describes ucsd-psystem-xc version 0.11
and was prepared 28 July 2012.

Table of Contents(ucsd-psystem-xc)

Table of Contents(ucsd-psystem-xc)

NAME

ucsd-psystem-xc − UCSD p-System Pascal cross compiler

DESCRIPTION

The *ucsd-psystem-xc* package is a collection of tools for compiling Pascal source files to produce UCSD p_System code files. The package includes:

ucsdpsys(1)
: A laucher to run the virtual machine comfortably from the command line. It includes a batch mode for automating (scripting) operations.

ucsdpsys_assemble(1)
: The cross assembler. It is able to assemble several different target microprocessor architectures in the one executable.

ucsdpsys_compile(1)
: The cross compiler. It understands the UCSD Pascal dialect, including UNIT definitions and references.

ucsdpsys_depends(1)
: May be used to determine include file dependencies, for use with *make*(1) and other build tools.

ucsdpsys_disassemble(1)
: For disassembling UCSD p-System code files. This is used to verify the correctness of the compiler.

ucsdpsys_downcase(1)
: A untility for converting Pascal code to lower case, leaving string constants and comments unaltered.

ucsdpsys_errors(1)
: A utility to translate back and forth between text and binary representations of the assembler error message files.

ucsdpsys_libmap(1)
: A utility for printing segment maps of UCSD p-System library files.

ucsdpsys_librarian(1)
: A utility for manipulating the segments within UCSD p-System codefiles.

ucsdpsys_link(1)
: A utility for linking UCSD p-System codefiles to their assembler components.

ucsdpsys_opcodes(1)
: A utility to translate back and forth between text and binary representations of the assembler opcode files.

ucsdpsys_setup(1)
: A utility to translate back and forth between text and binary representations of the `system.miscinfo` file.

Sister Projects

Some other projects will be of interest to you.

ucsd-psystem-fs
: This package contains tools for manipulating UCSD p-System floppy disk images, and a file system for mounting them in Linux as real file systems.
http://ucsd-psystem-fs.sourceforge.net/

ucsd-psystem-os
: This project provides a self-hosting set of system sources. You need the disk images produced by this project for the virtual machine to have a “system.pascal” file to run (this provides runtime support and the user command executive). This is a work in progress.

uvsd-psystem-vm
: This package provides a fully featured UCSD p-Machine emulator.

ARCHIVE SITE

The latest version of *ucsd-psystem-xc* is available on the Web from:

URL:	http://ucsd-psystem-xc.sourceforge.net/	
File:	ucsd-psystem-xc-0.11.README	# Description, from the tar file
File:	ucsd-psystem-xc-0.11.lsm	# Description, LSM format
File:	ucsd-psystem-xc-0.11.tar.gz	# the complete source
File:	ucsd-psystem-xc-0.11.pdf	# Reference Manual

BUILDING ucsd-psystem-xc

Full instructions for building *ucsd-psystem-xc* may be found in the *BUILDING* file included in this distribution.

COPYRIGHT

ucsd-psystem-xc version 0.11

It should be in the *LICENSE* file included with this distribution.

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

RELEASE NOTES

This section details the features and bug fixes of each of the releases.

Version 0.11 (2012-Jul-28)

- Kai Henningsen <kai.extern@gmail.com> discovered that 'Makefile' files generated by ucsdpsys_osmakgen did not correctly support the 'distclean' target. This has been corrected.
- Work is in progress to be able to cope with multiple p-machine versions.
- The compiler is now able to cope with variables declared in plain units.
- The *ucsdpsys_osmakgen*(1) command now understands how to generate the necessary debian/ files for building a debian package from the ucsd p-system operating system sources.
- Thw *ucsdpsys*(1) file no longer creates the implied system disk image if one of the supplied disk images is a functioning system disk.
- The *ucsdpsys*(1) command now better understands where ucsd-psystem-os installs its files, which it needs in order to build the default system disk image.

Version 0.10 (2011-May-18)

- A bug which caused a segfault in the *ubsdpsys −−batch* option has been fixed.
- The *ucsdpsys_osmakgen*(1) command, used by the ucsd-psystem-os project to generate its `Makefile`, now understands the presence of man pages, and installs them appropriately.

Version 0.9 (2011-Feb-02)

- The slides of the LCA 2011 talk "Factory Factory Factories" is now available in the web site.
- The *ucsdpsys_osmakgen*(1) command has been improved, with a view to Debian packaging of the OS.
- The *ucsdpsys*(1) command has a new **−−no−system** option, to suppress the construction of a system disk image.
- There is a new *ucsdpsys_compile*(1) option, **−−library−path** for adding directories to the library search path.
- The *ucsdpsys_compile*(1) command now fully supports the `(*$U` *filename* `*)` control comment.
- The *ucsdpsys_assemble*(1) command now understands the `.error` `.print` `.sbttl` `.title` pseudo-ops, mostly named for PDP-11 assmebler pseudo-ops of the same name.
- The *ucsdpsys_charset*(1) command has been moved to this project, out of the ucsd-psystem-fs project.
- The *ucsdpsys_assemble*(1) command now understands how to produce assembler listings, using the **−L** option. See *ucsdpsys_assemble*(1) for more information.
- The *ucsdpsys_compile*(1) command now issues warnings for unreachange statements. There is a new `(*$warning unreachable false *)` control comment to disable the warning.
- The project download web page now includes a link to the LunchPad PPA, where pre-compiled Ubuntu packages are available.
- The *ucsdpsys_assemble*(1) command now understands the `.ref` pseudo-op, and generates the appropriate relocation information.
- The *ucsdpsys_assemble*(1) command now more closely emulates the UCSD native assembler, in the way it forgets symbols created between one `.proc` and another. This stops historical source files from complaining about multiply defined symbols all over the place.
- The *ucsdpsys_assemble*(1) command now requires that the architecture be explicitly stated, either with the `.arch` pseudo-op, or the **−−arch** command line option, in all cases.
- The *ucsdpsys_assemble*(1) command now ignores all input after the `.end` directive.
- The *ucsdpsys_assembler*(1) command now understands `.gt` greater than, `>=` greater than or equal, `.lt` less than, `<=` less than or equal, `<>` inequality, and `=` equality comparisons.

- The *ucsdpsys_assemble*(1) command, now understands, for 6502 opcodes, how to relocate segment relative addresses for absolute addressing opcodes.
- The *ucsdpsys_assemble*(1) command now understands conditional assembly `.if`, `.else` and `.endc` pseudo-op directives.
- The *ucsdpsys_assembler*(1) command now understands the `.macro` pseudo-op, for defining an substituting macros into the code stream.
- A bug has been fixed in the code that checks codefiles for validity. It no longer rejects segment dictionaries with zero-length UNITSEG segments. These are produced when a program USES a non-intrinsic unit, but is not yet linked.

Version 0.8 (2010-Aug-28)

- The *ucsdpsys_assemble*(1) cross assembler now understands the `.func` pseudo-op.
- The error message formatting has been changed to use a 4 character hanging indent for multi-line error messages.
- A bug has been fixed in the *ucsdpsys_osmakgen*(1) command, it now correctly understands how to remove system segments from libraries with an assembler component.
- The *ucsdpsys_osmakgen*(1) command now understands how to link Pascal programs with their assembler components.
- A bug has been fixed in the *ucsdpsys*(1) command, it no longer fails if its temporary files are unlinked twice.
- There is a new *ucsdpsys_compile*(1) **−−view−path** option, symmetric with the *ucsdpsys_assemble*(1) and *ucsdpsys_depends*(1) commands' options of the same name.
- The *ucsdpsys_assemble*(1) command now understands the `.incude` pseudo-op. This is also a new corresponding **`-I`** command line option.
- A bug has been fixed in the *ucsdpsys_librarian*(1) command, it now patches the segment number in the procedure dictionary when it renumbers a segment.
- A bug has been fixed in the *ucsdpsys_disassemble*(1) and *ucsdpsys_libmap*(1) commands, they were printing SEPPROC link information incorrectly.
- The *ucsdpsys_osmakgen*(1) command now generates an "install" target, so that the results of the build can be installed into the system.
- The *ucsdpsys_assemble*(1) cross assembler now groks unary minus (−e) unary plus (+e) bit-wise and (e1 & e2), bit-wise or (e1 | e2), bit-wise not (~e), bit-wise exclusive-or (e1 ^ e2), and modulo (e1 % e2) expressions.
- The *ucsdpsys_compile*(1) cross compiler can now cope with VAR clauses in the IMPLEMENTATION section of a UNIT.
- The *ucsdpsys_compile*(1) cross compiler is now able to cope with units that export variables, noth intrinsic and non-intrinsic.
- The *ucsdpsys_compile*(1) grammar now understands "var anything" parameters to external assembler procedures and functions.
- The *ucsdpsys_osmakgen*(1) command now understands assembler source file include dependencies.
- The *ucsdpsys_depends*(1) command now understands how to process assembler source files, when looking for include dependencies.
- The *ucsdpsys_assemble*(1) command now procuces minimally correct relocation data sectiosn for each native code procedure. The *ucsdpsys_disassemble*(1) command now has a minimally correct understanding of relocation data.
- There is a new *ucsdpsys_link*(1) command, that may be used to link programs and libraries of separate procedures and functions together, to produce executable output codefiles. See *ucsdpsys_link*(1) for

more information.

- The *ucsdpsys_libmap*(1) and *ucsdpsys_disassemble*(1) commands now include the EOFMARK link information record, to be sure it contains the correct argument. The *ucsdpsys_assemble*(1) and *ucsdpsys_compile*(1) commands now correctly generate EOFMARK link information records.
- The *ucsdpsys_littoral*(1) command now correctly translates `nil` to `NULL`.
- The *ucsdpsys_littoral*(1) command now expands `with` variables completely. This preserves the semantics into the C++ code.
- There is now a build dependency on the *libexplain* project (http://libexplain.sourceforge.net/).
- A bug has been fixed in the *ucsdpsys*(1) command, it no longer overwrite its own temporary files. All of the *ucsdpsys*(1) options now have long versions as well. The UCSD p-System volumes that are created on-the-fly are now created large enough to hold all of the data.
- The *ucsdpsys_osmakgen*(1) command is now able to figure out when it needs to make a copy of `system/globals.text` based on include dependency information and the source file manifest.
- The `for` statement now understands `real` control variables. Note that the native compiler does not allow this.
- The *ucsdpsys_assemble*(1) cross assembler now understands the `.def` pseudo-op.

Version 0.7 (2010-Jun-21)

- There is a new *ucsdpsys_osmakgen*(1) command, used to write the `Makefile` for the ucsd-psystem-os project.
- The *ucsdpsys_setup*(1) command now accepts an `--`**arch** option, in order to select the byte sex of the `SYSTEM.MISCINFO` file it generates.
- There is a new *ucsdpsys_errors*(1) command, to translating the assembler error files from text to binary.
- The *ucsdpsys_opcode*(1) command now understands the opcode file format used by the UCSD Adaptive Assembler.
- A bug has been fixed in the *ucsdpsys_depends*(1) command, it no longer writes to a file called "−" when it should write to the standard output.
- The *ucsdpsys_librarian*(1) command has a new `--remove-system-segments` option, used to remove dummy segments from a `(*$U-*)` utility.
- The *ucsdpsys_librarian*(1) command is now able to renumber segments when they are transferred bwtween codefiles.
- The *ucsdpsys_compile*(1) command has a new `--`**host** option, that allows you to set the byte-sex based on the name of the host. Which helps those of us who don't necessarily remember what endian-ness all of the hosts actually are.
- The *ucsdpsys_assemble*(1) command has a new `--`**architecture** option, to permit the target architecture to be set from the command line.
- The *ucsdpsys_assemble*(1) multi-target cross assembler now has the beginnings of support for PDP-11 assembler.
- The cross compiler is now able to recognize the ord/odd hack (used to gain access to bit-wise opcodes) and turn such expression trees from logical operations into bit-wise operations.
- The disassembler no longer rejects valid machine code segments with very short procedures.
- The *ucsdpsys_assemble*(1) multi-target cross assembler now has beginnings of 6502 support, including both the MosTech syntax and the Apple syntax.
- A bug has been fixed in the cross compiler, it now generates the correct opcode for the inline-math sqrt function.

- The assembler now has a `.radix` pseudo-op, that may be used to change the default radix being used by the assembler.
- A bug has been fixed in repeat/until statements, it was generating no code in some cases.

Version 0.6 (2010-May-30)

- The compiler now understands EXTERNAL function and procedure declarations, and produces corresponding linker records.
- The compiler now has complete long integer support.
- The compiler now understands the built-in STR function.
- It is now possible to write long integer constants in the source code. They take the same forms as other integer constants, except they are suffixed with the letter L. This is an idea transplanted from C, the UCSD native compiler does not recognise such constants. It makes testing and debugging the long integer constant folding much easier.
- The compiler now understands `unit` definitions, using II.1 syntax and semantics. If II.0 `separate unit` definitions are seen, they result in a warning, and the `separate` keyword is otherwise ignored.
- The compiler now understands a C-style ternary operator expression `(e1 ? e2 : e3)`. The UCSD native compiler doesn't have this.

Version 0.5 (2010-May-17)

- There is a new `(*$feature underscore-significant true*)` contol comment, that may be used for increased ISO 10206 conformance.
- A bug has been fixed in the RECORD code, it no longer places the selector variable in the variant part of the record, and thus is no longer requesting memory from NEW that is one word short.
- There is a new `(*$feature efj-nfj false*)` control comment to turn off the use of the EFJ and NFJ opcodes.
- There is a new `(*$feature short-with false*)` control comment, that can be used to turn off WITH statement optimizations.
- The built-in UNITWRITE procedure now accepts string constants for the second parameter. The UCSD native compiler did not allow this. Handy for debugging the system I/O procedures.
- The compiler now optimizes IF stratements with GOTO clauses. It now goes directly to the label from the condition, when possible, rather than using UJP in the individual clauses.
- The IF statement now generates better code for the case where THEN is empty but ELSE is not.
- The compiler now understands the ISO 10206 integer constants with an explicit radix. This was not available in the UCSD native compiler, for obvious reasons.
- The is a new *ucsdpsys_setup*(1) command, used to encode and decode the `SYSTEM.MISCINFO` file.
- There is a new *ucsdpsys_downcase*(1) command, that may be used to convert identifiers in Pascal source code from upper case to lower case.
- The compiler no longer has a problem with sets passed as parameters. The way sets are push onto the stack has been further optimized.
- The compiler now understands how to optimize away MOVELEFT, MOVERIGHT and FILLCHAR with a constant zero or negative length.
- A bug has been fixed in the IN operator, in the case where the set had a fixed size.
- A bug has been fixed in the constant folding of string comparisons, it was getting relational comparisons (<, <=, >, >=) wrong, but equality comparisons (=, <>) right.
- A bug has been fixed in the indexing of byte arrays (pointers) with enum types. It no longer throws an assert.

- The compiler now issues warnings for comments that are not ISO 7185 comforming.
- A bug has been fixed in the code generation of MOV opcodes, in the case where more than 127 words had to be moved.
- The compiler now understands `arctan` (ISO 10206) as a synonym for `atan`, but only if `(*$feature inline-math true*)` is in effect.
- The compiler now generates correct code for NOT logical expressions assigned to a boolean variable, or passed as a boolean parameter.
- A bug has been fixed in the code that folds constant MPI (integer multiply) expressions.
- A bug has been fixed in the optimization of integer subtraction.
- A bug has been fixed in the optimization of the ADI (add integer) expression.
- A bug has been fixed in the optimisation of the logical NOT expression.
- The cross compiler now understands the bit-wise integer AND, OR and NOT expressions.
- The compiler now generates LDB (load byte) and STB (store byte) instructions for packed arrays of 8-bit things, not just packed array of char. This is the same behaviour as the UCSD native compiler.
- There is a new *ucsdpsys_librarian*(1) **−R** option, that can be used to remove segments by name or by number.

Version 0.4 (2010-May-06)

- A bug has been fixed in the code generation for large set constants.
- The CASE statement now understands negative case values.
- The compiler now understands how to cast string constants into packed-array-of-char constants, when they are procedure and functions parameters.
- The compiler now understands when a case control expression is a function call with no parameters.
- The compiler now understands functions calls with no parameters on either side of the IN operator.
- The compiler now generates the correct code for segment procedures that are declared forward.
- The compiler now understands how to pass parameters that are records, by value.
- The compiler now generates correct code for array parameters when they are passed by value.
- A bug has been fixed in the READLN code generation, it no longer throws an assert.
- The compiler no longer issues syntax errors when semicolons appear in questionable places in RECORD declarations.
- The way symbol conflicts and shadows are calaculated has been changed, it was getting false positive on the conflict tests.
- The compiler now understands passing a string as the first parameter to the FILLCHAR procedure.
- The compiler now understands the unary plus operator.
- The compiler now understands the built-in GET, GOTOXY, PAGE, PUT, SEEK, UNITSTATUS and UNITWAIT procedures.
- There is a new (*$feature inline-math true*) control comment. When this is enabled, the compiler now understands the built-in ATAN, COS, EXP, LN, LOG, SIN and SQRT functions.
- There is a new *ucsdpsys_assemble*(1) command, that may be used to assemble machine code and p-code. It isn't particularly capable, as yet, but it will become more so as work proceeds on the p-machine validation
- The compiler now accepts for loops of char values where one or both limits are char constants.
- The built-in FILLCHAR procedure now accepts its third paramater being an enumerated type. This is for backwards compatibility with the UCSD native compiler.

- The compiler now understands how to index an array by a char value. Previously it was throwing an assert.
- There is a new (*$feature ignore-undefined-segment-zero true *) option, that can be used to turn off checking for undefined forward declarations, when those symbols would be in segment zero. This “feature” is used by system utilities. All other cases of forward functions being undefined result in a fatal error; use EXTERNAL for procedures to be linked later.
- The disassembler can now cope with broken pointers in a segment’s procedure dictionary. Usually undefined (external) procedures with have a zero (0) entry in the procedure table.
- The string parameters length check is now a warning, rather than an error. This is because the implicit copy at run-time will throw a run-time error of the string doesn’t fit.
- The compiler now accepts calls to the built-in EOF and EOLN functions with no parameters.
- The code generation for empty set constant has been improved. It no longer throws an assert. The same assert revealed that empty sets as a function parameter was not correctly being cast to the appropriate type of set.

Version 0.3 (2010-Apr-25)

- A warning is now issued if a case statement contains an `otherwise` clause. You can disable the warning by using the `(*$warning otherwise false*)` control comment.
- The compile listing now includes the symbol table for each procedure and function.
- A bug has been fixed in the code that derefereces pointers to strings. It no longer tries to laod the whole string onto the stack. The compiler now understands how to deal with string-typed fields on the right hand side of dot (expr.name) expressions.
- A bug has been fixed where function parameters that were the names of functions that had no parameters were not being called.

The compiler no longer issues duplicate label warnings. In some cases it was issuing warnings about unused labels twice.

- The compiler now understands the built-in COPY, DELETE, EOF, EOLN, FILLCHAR, INSERT, POS, UNITBUSY and UNITCLEAR functions and procedures.
- The compiler no loger throws an assert if a procedure in segment zero is EXIT()ed.
- The compiler now correctly scopes enumerated constant definitions that are declared within the record scopes.
- A bug has been fixed in the code that copied non-var string parameters into their local temporaries.
- Thw compiler now understands how to perform a non-local function return assignment.
- The compiler now also accepts an integer value as the third parameter of fillchar, even thouh it is documented to take a char value.
- A bug has been fixed where constant negative array indexes would cause an assert to fail. It turned out that some optimizations were not checking the range of offsets, and creating invalid offsets.
- The compiler now understands declaring and accessing arrays using multi dimension syntax.
- A number of error messages concerning forward declared types have been improved; they are now earlier, and less cryptic.
- A bug has been fixed in the code generation of constant sets. They are no longer all-bits-zero, but instead contain the correct value.
- The compiler now only range checks the CHR parameter if requested. The UCSD native compiler did not range check CHR.
- The compiler now checks parameter string lengths (declared *vs* actual) for overruns.

- The compiler now understands about fileˆ variables.
- The *ucsdpsys*(1) command is now better at cleaning up its temporary files.
- The boolean comparison operators (=, <>, <=, <, >=, >) now have additional code to cope with one side or the other being a constant.
- A bug has been fixed in the way constant folding was handled around the FOR statement's limits.

Version 0.2 (2010-Apr-19)

- The target for this release was to be able to compile the UCSD native Pascal compiler from source. This has been achieved. It has yet to be determined if the compiler thus created actually functions.
- For differences between this cross compiler and the UCSD native compiler, see the *ucsdpsys_compile*(1) man page. The most notable difference is that SIZEOF is a keyword, requiring the UCSD native compiler's PROCEDURE SIZEOF to be renamed.
- Numerous bugs have been fixed, usually in unexplored corner cases.
- The compiler now understands the ABS, BLOCKREAD, BLOCKWRITE, CLOSE, CONCAT, EXIT, HALT, IDSEARCH, IORESULT, KEYBOARD, LENGTH, MARK, MOD, MOVELEFT, MOVERIGHT, OPENNEW, OPENOLD, PWROFTEN, READ, READLN, RELEASE, RESET, REWRITE, ROUND, SCAN, TREESEARCH, TRUNC, UNITREAD, UNITWRITE and WRITELN built-in symbols.
- The STRING type has been turned into a built-in named type. This permits the unwise user to redefine STRING to be a variable or a procedure or a function, or (for maximum confusion) a different type. This is what shadow warnings are for.
- The compiler now understands the CASE, FOR, REPEAT UNTIL and WITH statements.
- The compiler now understands comparisons of CHAR values.
- The compiler now accepts pointers as parameters to the ORD function. This seems oddly inconsistent, in a language as intent as Pascal is, with the protection of the programmer from his own folly.
- The compiler now understands set arithmetic and set comparisons.
- It is now possible, using the *ucsdpsys_compile* −−*listing* option, to obtain a compiler listing. The listing contains the source code interleaved with the disassembled p-code. The `(*$L)` control comment is ignored.
- The compiler now understands = and <> comparisons of multi-word values (arrays and records).
- The compiler can now be configured to have longer identifier (name) lengths. It defaults to 8 for compatibility, and it still drops underscores.
- The compiler now understands comparisons of packed arrays of char.

Version 0.1 (2010-Apr-01)

First public release.

- The following built-in functions are understood: CHR, MEMAVAIL, ODD, ORD, PRED, SIZEOF, SQR, SUCC, TIME.
- All of the usual Pascal expresion operators are understood, although not always across the full range of parameter types.
- The cross compiler can produce both little-endian codefiles and big-endian codefiles.
- A number of features from modern Pascal implementations are avilable: hex constants, binary constants, short-circuit boolean evaluation, the address-of (@) operator,
- Most of the Pascal statement types are available, including: BEGIN END, CASE (and OTHERWISE), FOR, GOTO (local), IF THEN (ELSE), NEW (including variant types), REPEAT UNTIL, WHILE, WITH, WRITE, WRITELN. It is not yet possble to use non-local GOTO.
- Segment procedures can be created, and UNIT interfaces can be accessed from library codefiles. It is not yet possible to compile UNITs. While FORWARD procedures and functions are understood, EXTERNAL procedures and functions are not yet supported.
- All of the UCSD Pascal data types are supported: ARRAY (including PACKED ARRAY), BOOLEAN, CHAR, enumerated, FILE, INTEGER INTERACTIVE, pointers, REAL, RECORD (including PACKED RECORD), SET, STRING (including STRING[n]), subrange, TEXT. The long integer types are not yet supported.
- The cross compiler understands many of the UCSD Pascal constants, including: FALSE, MAXINT, NIL, TRUE,
- The cross compiler is able to optimize most statements and expressions better than the Apple Pascal native compiler. Constant expressions are folded at compile time.
- There is a *ucsdpsys_depends*(1) command, that can be used by your build system to scan for `(*$I filename*)` include directives.

Version 0.0 (2006-May-22)

No public release.

NAME

How to build ucsd-psystem-xc

BEFORE YOU START

There are a few pieces of software you may want to fetch and install before you proceed with your installation of ucsd-psystem-xc.

Boost Library

You will need the C++ Boost Library. If you are using a package based system, you will need the *libboost-devel* package, or one named something very similar.
http://boost.org/

libexplain

The *ucsd-psystem-xc* package depends on the libexplain package: a library of system-call-specific strerror replacements.
http://libexplain.sourceforge.net/

GNU Groff

The documentation for the *ucsd-psystem-xc* package was prepared using the GNU Groff package (version 1.14 or later). This distribution includes full documentation, which may be processed into PostScript or DVI files at install time − if GNU Groff has been installed.

SITE CONFIGURATION

The **ucsd-psystem-xc** package is configured using the *configure* program included in this distribution.

The *configure* shell script attempts to guess correct values for various system-dependent variables used during compilation, and creates the *Makefile* and *lib/config.h* files. It also creates a shell script *config.status* that you can run in the future to recreate the current configuration.

Normally, you just *cd* to the directory containing *ucsd-psystem-xc*'s source code and then type

```
% ./configure
...lots of output...
%
```

Running *configure* takes a minute or two. While it is running, it prints some messages that tell what it is doing. If you don't want to see the messages, run *configure* using the quiet option; for example,

```
% ./configure --quiet
%
```

To compile the **ucsd-psystem-xc** package in a different directory from the one containing the source code, you must use a version of *make* that supports the *VPATH* variable, such as *GNU make*. Change directory to the directory where you want the object files and executables to go and run the *configure* script. The *configure* script automatically checks for the source code in the directory that *configure* is in and in .. (the parent directory). If for some reason *configure* is not in the source code directory that you are configuring, then it will report that it can't find the source code. In that case, run *configure* with the option `--srcdir=`*DIR*, where *DIR* is the directory that contains the source code.

By default, *configure* will arrange for the *make install* command to install the **ucsd-psystem-xc** package's files in */usr/local/bin*, and */usr/local/man*. There are options which allow you to control the placement of these files.

`--prefix=`*PATH*

This specifies the path prefix to be used in the installation. Defaults to */usr/local* unless otherwise specified.

`--exec-prefix=`*PATH*

You can specify separate installation prefixes for architecture-specific files files. Defaults to *${prefix}* unless otherwise specified.

`--bindir=`*PATH*

This directory contains executable programs. On a network, this directory may be shared between machines with identical hardware and operating systems; it may be mounted read-only.

Defaults to *${exec_prefix}/bin* unless otherwise specified.

`--mandir=`*PATH*
: This directory contains the on-line manual entries. On a network, this directory may be shared between all machines; it may be mounted read-only. Defaults to *${prefix}/man* unless otherwise specified.

The *configure* script ignores most other arguments that you give it; use the `--help` option for a complete list.

On systems that require unusual options for compilation or linking that the *ucsd-psystem-xc* package's *configure* script does not know about, you can give *configure* initial values for variables by setting them in the environment. In Bourne-compatible shells, you can do that on the command line like this:

```
$ CXX='g++ -traditional' LIBS=-lposix ./configure
...lots of output...
$
```

Here are the *make* variables that you might want to override with environment variables when running *configure*.

Variable: CXX
: C++ compiler program. The default is *c++*.

Variable: CPPFLAGS
: Preprocessor flags, commonly defines and include search paths. Defaults to empty. It is common to use `CPPFLAGS=-I/usr/local/include` to access other installed packages.

Variable: INSTALL
: Program to use to install files. The default is *install* if you have it, *cp* otherwise.

Variable: LIBS
: Libraries to link with, in the form `-l`*foo* `-l`*bar*. The *configure* script will append to this, rather than replace it. It is common to use `LIBS=-L/usr/local/lib` to access other installed packages.

If you need to do unusual things to compile the package, the author encourages you to figure out how *configure* could check whether to do them, and mail diffs or instructions to the author so that they can be included in the next release.

BUILDING UCSD-PSYSTEM-XC

All you should need to do is use the

```
% make
...lots of output...
%
```

command and wait. When this finishes you should see a directory called *bin* containing several programs.

If you have GNU Groff installed, the build will also create a *etc/reference.ps* file. This contains the README file, this BUILDING file, and all of the man pages.

You can remove the program binaries and object files from the source directory by using the

```
% make clean
...lots of output...
%
```

command. To remove all of the above files, and also remove the *Makefile* and *lib/config.h* and *config.status* files, use the

```
% make distclean
...lots of output...
%
```

command.

The file *etc/configure.in* is used to create *configure* by a GNU program called *autoconf*. You only need to know this if you want to regenerate *configure* using a newer version of *autoconf*.

TESTING UCSD-PSYSTEM-XC

The *ucsd-psystem-xc* package comes with a test suite. To run this test suite, use the command

```
% make sure
...lots of output...
Passed All Tests
%
```

The tests take a few seconds each, with a few very fast, and a couple very slow, but it varies greatly depending on your CPU.

If all went well, the message

```
Passed All Tests
```

should appear at the end of the make.

INSTALLING UCSD-PSYSTEM-XC

As explained in the *SITE CONFIGURATION* section, above, the *ucsd-psystem-xc* package is installed under the */usr/local* tree by default. Use the `--prefix=`*PATH* option to *configure* if you want some other path. More specific installation locations are assignable, use the `--help` option to *configure* for details.

All that is required to install the *ucsd-psystem-xc* package is to use the

```
% make install
...lots of output...
%
```

command. Control of the directories used may be found in the first few lines of the *Makefile* file and the other files written by the *configure* script; it is best to reconfigure using the *configure* script, rather than attempting to do this by hand.

GETTING HELP

If you need assistance with the *ucsd-psystem-xc* package, please do not hesitate to contact the author at

```
Peter Miller <pmiller@opensource.org.au>
```

Any and all feedback is welcome.

When reporting problems, please include the version number given by the

```
% ucsdpsys_compile -V
ucsdpsys_compile version 0.11.D001
...warranty disclaimer...
%
```

command. Please do not send this example; run the program for the exact version number.

COPYRIGHT

ucsd-psystem-xc version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

Factory factory factories − Abandon all flow of control Ye who enter here

ABSTRACT

In many cases, allegedly OO code is still highly procedural and imperative, with little advantage taken of the possibilities presented by inheritance and virtual methods. This talk is about delegating flow of control to an unknown future, manufacturing objects that in turn manufacture more objects, of various class relationships. Why is this useful? How do you follow the program logic, especially if the classes haven't even been written yet? How come the combinatorial explosion doesn't make it untestable? Come along and take a trip down the factory**n rabbit hole, a warren several layers deep, inside a compiler.

INTRODUCTION

There is a particular technique used in the ucsd-psystem-xc project to construct and manipulate Abstract Syntax Tree (AST) representations of the Pascal program. Rather than having the tree operations be implemented by procedural code external to the tree, the manipulations are performed by the tree nodes themselves.

A design goal was to be able to re-use the grammar for the Pascal language, so that other static analysis tools could also be written, but having the grammar and symbol table handling remain in common library code. This complicates things, if we are going to have the tree nodes performing all the work, because this would seem to imply that every tree node would include the methods necessary to perform all tasks and re-uses of the grammar. Happily, this is not the case.

This paper is an extension of the earlier *Compilers and Factories* paper.

THE VIRTUAL KEYWORD

The key concept here is the `virtual` keyword in C++. A virtual method is one that can have different implementations in different derived classes. Thus, for our AST node to perform a different operation, it must be a different derived class.

Some Revision

Long, long ago, there was no C++. Examples of AST representations dating from then would often have C declarations like this:

```
struct expr_t
{
    int kind;
    union
    {
        int value;
        struct
        {
            struct expr_t *lhs;
            struct expr_t *rhs;
        } p;
    } u;
};
```

Manipulating these trees would involve a function such as this:

```
int
expr_evaluate(const struct expr_t *ep)
{
    switch (ep->kind)
    {
    case CONSTANT:
        return ep->u.value;

    case PLUS:
        return expr_evaluate(ep->u.p.lhs)
            + expr_evaluate(ep->u.p.rhs);
```

```
            case MINUS:
                return expr_evaluate(ep->u.p.lhs)
                    - expr_evaluate(ep->u.p.rhs);

            etc
            }
        }
```

Each time you wanted to add a new kind of expression node, you had to visit each of these functions, and add another switch case. This can become an expensive maintenance problem, and also lead to version control bottlenecks for the development team.

In order to be able to add code in the future, but not have these problems, it is necessary to split the problem into pieces, using pointers to functions:

```
expr_evaluate(const struct expr_t *ep)
{
    return (*ep->evaluate_method)(ep);
}
```

This means our struct declaration changed as well

```
struct expr_t
{
    int (*evaluate_method)(const struct expr_t *ep);
    union
    {
        int value;
        struct
        {
            struct expr_t *lhs;
            struct expr_t *rhs;
        } p;
    } u;
};
```

Notice, in particular, that the `kind` member is now gone, replaced by one or more function pointers. In practice, this tends to be a pointer to a struct full of function pointers, one for each task, because this simplifies the creating of new AST nodes.

All of which means that our actual evaluation comes in separate pieces:

```
int
expr_constant_evaluate(const struct expr_t *ep)
{
    return ep->u.value;
}

int
expr_plus_evaluate(const struct expr_t *ep)
{
    return expr_evaluate(ep->u.p.lhs)
        + expr_evaluate(ep->u.p.rhs);
}
```

The actual implementation would have these in separate compilation units. Now that we have split this up, it would also be possible to do away with the union, and `malloc` AST nodes of the appropriate size.

If anyone has done this manually, you will know that there is a lot of machinery that needs to be kept in sync. Much of this machinery is done for you by C++, and it also adds some rigor to the types of nodes, avoiding the numerous type casts required when doing the same thing manually. The C++ could would look something like this:

```
class expression
{
public:
    virtual int evaluate(void) const = 0;
};
```

and the implementations

```
class expression_plus:
    public expression
{
public:
    int
    evaluate(void)
        const
    {
        return lhs->evaluate() + rhs->evaluate();
    }

private:
    expression *lhs;
    expression *rhs;
};
```

The key thing to notice is that we replaced the `kind` member with a “vtable”, and switches on `kind` with virtual methods.

Flow Of Control

Once all of the machinery is in place, adding a new kind of expression AST node simply means deriving a new class, and implementing the appropriate methods, such as *evaluate* in the above example. If you are a new developer on the team, and you didn’t see the machinery unfold, and implemented the first few classes, just how the code actually *reaches* your virtual method can be a bit of a mystery.

The first thing to remember is what a `virtual` method is. It is a type-based dispatch mechanism. There many only be a single call to that method in the entire program, and yet there could be tens or hundreds of implementations of that method. There is no voodoo here, no magic. If it were done long-hand, as in the first example, confusion rarely arises. Just think of it as the same thing, only distributed differently amongst the source files.

The second thing to remember is that you often *don’t care* how the code is called, because that mechanism has already been debugged. When flow of control does get to you, all you care about is getting your bit right.

Testability

Is using a virtual method inherently more difficult to test than the original C implementation? They both have the same code, doing the same jobs, the code is merely distributed amongst the source files differently. So, no, the testing burden is unchanged. Do not mistake the C++ verbosity for “more stuff to test”, and remember that C++ is *very* verbose.

Quite possibly, the separation of functionality by class means that you can have greater confidence that you will not unintentionally break something else in the file, because you are not even editing the same files.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/expression.h` file, and its derived classes may be found in the `lib/expression/`*`derived`*`.h` and *`tool`*`/expression/`*`derived`*`.h` files (the directory hierarchy mirrors the class hierarchy). The parser can be found in the `lib/pascal/grammar.y` file.

THE FACTORY CONCEPT

A factory in this sense is a function that returns new instances of a class. Think of a parser that reads text, parses it into expressions, and returns a pointer to the abstract syntax tree representing the parsed

expression. This is an example of a factory.

Imagine that our (vastly simplified) yacc grammar looked like this:

```
expr
    : NUMBER
        {
            $$ = constant_expr_factory($1);
        }
    | IDENTIFIER
        {
            $$ = name_expr_factory($1);
        }
    | expr '+' expr
        {
            $$ = plus_expr_factory($1, $3);
        }
    | expr '-' expr
        {
            $$ = minus_expr_factory($1, $3);
        }
    ;
```

For each kind of expression, we have a factory that can build them for us. They are not especially complicated:

```
expr *
constant_expression_factory(int value)
{
    return new expr_constant(value);
}
```

But why wouldn't we just put the same code into the grammar production {rules}? Because we wanted to re-use the grammar.

VIRTUAL FACTORIES

The grammar can be re-used by more than one translation task if we add a context object, and some virtual methods:

```
expr
    : NUMBER
        {
            $$ = ctx->constant_expr_factory($1);
        }
    | IDENTIFIER
        {
            $$ = ctx->name_expr_factory($1);
        }
    | expr '+' expr
        {
            $$ = ctx->plus_expr_factory($1, $3);
        }
    | expr '-' expr
        {
            $$ = ctx->minus_expr_factory($1, $3);
        }
    ;
```

And the `ctx` variable is a pointer to

```
class translator
```

```
{
public:
    virtual expr *constant_expr_factory(int) = 0;
    virtual expr *name_expr_factory(int) = 0;
    virtual expr *plus_expr_factory(expr *, expr *) = 0;
    virtual expr *minus_expr_factory(expr *, expr *) = 0;
    etc
};
```

By deriving different `translator` classes, we can have one translator that implements a compiler, one that implements a pretty printer, one that calculates cyclomatic complexity statistics, *etc*.

The compiler translator creates expression tree nodes that have an implementation that compiles the expressions. The pretty printer translator creates different expression tree nodes that have an implementation that prints the expressions out. And other static analysis tools each have their own implementations.

Testability

Does this make programs that use this technique harder to test? The amount of code to be written is the same, and does the same jobs. So, no, the testing burden is unchanged.

However, you have the advantage that the parser is common to all of the tools, and so bug fixes to the parser are inherited by all tools. Change once, test everywhere? Not quite: if you had *n* separate yacc files, all with the same bug, you would have to make *n* identical changes, and re-test *n* tools. Testing burden unchanged, *but* the probability of unintentionally diverging grammars becomes zero.

Flow Of Control

The need to understand the flow of control comes when the developer is testing a new derivation of the `translator` class. The grammar, and its connection to the translator context has already been written and tested, all you need to do is test the newly derived class. Your test cases, then, must exercise each of the new factory methods, one test for each of the expression productions, and flow of control will then enter each of the factory methods.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/translator.h` file, and its derived classes may be found in the *`tool`*`/translator/`*`derived`*`.h` files.

FACTORY FACTORIES

The wheels of this context concept would appear to start to come off when we consider assignment expressions. A grammar for a C-like language could look like this:

```
expr
    : IDENTIFIER
        {
            $$ = ctx->name_expr_factory($1);
        }
    | expr '=' expr
        {
            $$ = ctx->assignment_expr_factory($1, $3);
        }
    | expr '+' expr
        {
            $$ = ctx->plus_expr_factory($1, $3);
        }
    ;
```

How does our name expression factory know which side of the assignment it is on? At code generation time, should it emit a load opcode or a store opcode? We don't know... yet. What we do know is that loads are much more likely than stores, so we initially generate expression trees that would perform loads.

But this just pushes the problem into the `assignment_expr_factory` method. In order to figure out

what kinds of assignment opcode to use, it would be necessary to figure out what kind of load opcode is present, and generate the corresponding store

```
expression *
translator_compiler::assignment_expr_factory(expression *e1, expression *e2
{
    const expr_load *test1 =
        dynamic_cast<const expr_load *>(e1);
    if (test1)
        return new expr_store(e1->get_operand(), e2);

    const expr_array_load *test2 =
        dynamic_cast<const expr_array_load *>(e1);
    if (test2)
        return new expr_store_array(e1->get_lhs(), e1->get_rhs(), e2);

    yyerror("inappropriate assignment");
    return new expression_error();
}
```

This makes me cringe. Those down-casts have my alarm bells going off. And all those getters so that AST node privates can be groped, ugh! But what alternative is there? To answer that, let's backtrack for a moment. Our very first example can be re-written like this:

```
int
expr_evaluate(const struct expr_t *ep)
{
    if (ep->kind == CONSTANT)
        return ep->u.value;

    if (ep->kind == PLUS)
        return expr_evaluate(ep->u.p.lhs)
            + expr_evaluate(ep->u.p.rhs);

    if (ep->kind == MINUS)
        return expr_evaluate(ep->u.p.lhs)
            - expr_evaluate(ep->u.p.rhs);

    etc
}
```

The chain of `if` statements in `assignment_expr_factory` is a *`switch` in disguise*, a type-based dispatch in disguise. We should be using a virtual method instead.

But in which class should we place the virtual method? Clearly, it isn't inside the `translator` class, since we tried it there already. The type-based dispatch is based on the expression type, and that is where the virtual method lives, in the `expression` class:

```
expr: expr '=' expr
    {
        $$ = $1->assignment_expr_factory($3);
    }
```

No, no, no, that can't be right: the `ctx` object doesn't get any chance to intervene. Except that it does: when it created the left hand side in the first place.

By creating, say, a compiler specific "load" AST node, it also created the assignment factory; they are the same object. There is no way a pretty printer assignment object will ever be created by a compiler load object (unless you deliberately code it that way).

Note, too, that the error-prone down-casts are *gone*, as is the need to grope anyone's privates. And the code

is faster, too, by eliminating the slow down-casts and multiple tests.

The sharp-eyed reader will have noticed that we have omitted the error case. What happens when it goes wrong? The easiest way is to have the common base class aways emit an error complaining about an inappropriate assignment, unless overridden.

```
expression *
expression::assignment_expr_factory(expression *, expression *)
{
    yyerror("inappropriate assignment");
    return new expression_error();
}
```

In summary, our `name_expr_factory` method manufactured an object that, in turn, contains an `assignment_expr_factory` method, used to manufacture more AST nodes. We now have a factory factory.

Testability

My head is starting to explode. Surely *now* there are combinatorial effects on testing!

Well, yes and no. Yes, programming languages by definition are capable of combinatorial effects when it comes to all the ways you can put together different expressions to build different programs; that is unchanged, compilers need *lots* of testing.

And, no, the factory factories do not making the testing burden worse. They are, after all, implementing the same thing, often with the same code, albeit distributed differently amongst the classes.

Flow Of Control

If I'm a developer adding a new type of assignment to an existing complier implemented this way, how to I know when execution will reach my shiny new expression class' `assignment_expr_factory` method? Well, the same way you would have when it was imperative code: write a test with that kind of assigment in it, and hand it to the parser. Remember: you aren't testing the parser part of the code, only your new assignment type (class).

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/expression.h` file, and its tool-specific derived classes may be found in the *`tool`*`/expression/`*`derived`*`.h` files.

FACTORY FACTORY FACTORIES

Now we turn our attention to the `name_expr_factory` method. It's been trying to look all innocent and inconspicuous.

```
expression *
translator_compiler::name_expr_factory(const char *name)
{
    symbol *sp = lookup(name);
    if (!sp)
    {
        yyerror("name unknown");
        return new expr_error();
    }

    const symbol_extern *test1 =
        dynamic_cast<const symbol_extern *>(sp);
    if (test1)
        return new expr_load_extern(sp);

    const symbol_static *test2 =
        dynamic_cast<const symbol_static *>(sp);
    if (test2)
        return new expr_load_static(sp);
```

```
        const symbol_local *test3 =
            dynamic_cast<const symbol_local *>(sp);
        if (test3)
            return new expr_load_local(sp);

        yyerror("can't use name here");
    }
```

This is another example of a type-based dispatch in disguise. But where does the virtual method belong? Clearly, not in the `translator` class or derivative, we already tried that. Instead, we implement it in the `symbol` class, as follows:

```
expression *
translator::name_expr_factory(const char *name)
{
    symbol *sp = lookup(name);
    if (!sp)
    {
        yyerror("name unknown");
        return new expr_error();
    }
    return sp->name_expr_factory();
}
```

We moved the `name_expr_factory` into the `translator` base class, because it is now identical across all derived classes, because it no longer needs to know about compiler-specific classes.

As in the previous section about assignment expressions: doing symbol accesses this way means that the advantages are the same, the testing burden unchanged, and the error handling is the same.

In summary, the `translator::name_expr_factory` method looked up a `symbol` object that, in turn, contains a `name_expr_factory` method, used to manufacture `expression` AST nodes, that in turn contain `assignment_expr_factory` methods, used to manufacture more `expr` AST nodes. We now have a factory factory factory.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/symbol.h` file, and its derived classes may be found in the `lib/symbol/`*`derived`*`.h` and *`tool`*`/symbol/`*`derived`*`.h` files.

FACTORY**4

Have you thought about variable scopes in Pascal? By having different scopes for `program`s and `function`s (because their variables are accessed by different opcodes) when a new variable is declared, you ask the current `scope` to manufacture a new `symbol` instance that... you get the idea.

The Source Code

This concept may be found the the *ucsd-psystem-xc* source code in the `lib/scope.h` file, and its derived classes may be found in the `lib/scope/`*`derived`*`.h` and *`tool`*`/scope/`*`derived`*`.h` files.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys − UCSD p-System launcher

SYNOPSIS

ucsdpsys [*option*...]
ucsdpsys −V

DESCRIPTION

The *ucsdpsys* program is used to start an instance of the UCSD p-System virtual machine, providing it with the necessary system volumes to function. This is done by wrapping the *ucsdpsys_vm*(1) command, possibly performing disk image operations before and after via the *ucsdpsys_disk*(1) command.

OPTIONS

The following options are understood:

−a

−−apple Execute the virtual machine in Apple compatibility mode, initialized using the same addresses as the original Apple][p-System.

−b *filename*

−−batch=*filename*
Start the virtual machine in batch mode. Input is read from *filename*, output is written to the standard output (unless **−x** is used). If "−" is specified, the standard input is used.

−f *filename*

−−forget=*filename*
Access the *filename* as a read-write disk image, but changes to the disk image are never written back to the file and are forgotten when the virtual machine stops. If a directory is specified, a temporary disk image is created with *ucsdpsys_mkfs*(1), and the contents of the directory are read into it before the virtual machine starts.

−g

−−debug
Run the virtual machine in debug mode.

−N

−−no−system
This option may be used to avoid constructing a system disk image. For the system to boot, you must provide a system disk using one of the **−r** or **−f** or **−w** options. Any **−S** options will be ignored.

−n *filename*

−−name=*filename*
Uses the given *filename* as the executable to launch. Defaults to `SYSTEM.PASCAL` if not specified. This file may be on any of the volumes, although it is traditional for it to be on the `SYSTEM:` volume.

−P *release-name*

−−p−machine=*release-name*
This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to "II.1" if not set.

−r *filename*

−−read=*filename*
Access the *filename* as a read-only disk image. If a directory is specified, a temporary disk image is created, and the contents of the directory are read into it before the virtual machine starts.

−S *directory*

−−system=*directory*
: This option give a directory to be searched for system files. This option may be given more than once. If this option is not given, the files installed by the *ucsd-psystem-os* and *ucsd-psystem-vm* packages will be used, if present.

−T

−−trace Trace execution of opcodes by the virtual machine.

−t *filename*

−−trace−file=*filename*
: Write the execution trace to the given file. If "−" is given as the *filename*, the trace will be written on the standard output.

−V

−−version
: Print the version of the *ucsdpsys* program being executed.

−w *filename*

−−write=*filename*
: Access the *filename* as a read-write disk image. If a directory is specified, a temporary disk image is created, and the contents of the directory are read into it before the virtual machine starts, and then read back out into the directory when the virtual machine stopes.

−x

−−xterm
: Start an *xterm*(1) for `CONSOLE:` and `SYSTERM:`. Especially useful when using the debugger and its messages are output to the standard output and stderr.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys* command will exit with a status of 1 on any error. The *ucsdpsys* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_vm(1)
: UCSD p-System virtual machine

ucsdpsys_disk(1)
: manipulate UCSD p-System disk images.

ucsdpsys_mkfs(1)
: create UCSD p-System disk images.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_assemble − UCSD p-System cross assembler

SYNOPSIS

ucsdpsys_assemble [*option*...] *filename*

ucsdpsys_assemble −VERSion

DESCRIPTION

The *ucsdpsys_assemble* program is used to assemble low-level machine source code into UCSD p-System code files. The result is not executable, it must be linked to a program in order to be executed.

OPTIONS

The following options are understood:

−A *name*

−−host=*name*
: This option may be used to specify the machine architecture from the command line, as if there was a `.arch "`*name*`"` pseudo opcode at the start of the source file. There is an equivalent option for the *ucsdpsys_compile*(1) command.

−I *directory*

−−include=*directory*
: This option is used to specify an include file directory to search. This option may be given more than once.

−J *directory*

−−view-path=*directory*
: This option is used to specify a directory to append to the view path. This option may be given more than once.

−L *filename*

−−listing=*filename*
: This option may be used to nominate a file to take the assembler listing. For each line of source text the listing includes the address of the opcodes (in hex), the data bytes of the opcodes (in hex), and the corresponding source line.

−o *filename*

−−output=*filename*
: This option may be used to specify the code file the result s are written to. If not specified, the extension is removed form the source file (if any) and a "`.code`" extension is added.

−P *release-name*

−−p−machine=*release-name*
: This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to "II.1" if not set.

−V

−−version
: Print the version of the *ucsdpsys_assemble* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_assemble* command will exit with a status of 1 on any error. The *ucsdpsys_assemble* command will only exit with a status of 0 if there are no errors.

EXPRESSIONS

This section details the arithmetic expressions understood by the assembler. In general, these are the same expression forms used by the UCSD native assembler; any exceptions will be noted.

Addition
: You can add two integer values using the usual plus (*e1* “+” *e2*) operator.

Subtraction
: You can subtract two integer values using the usual minus (*e1* “−” *e2*) operator.

Multiplication
: You can multiply two integer values using the usual star (*e1* “*” *e2*) operator.

Division
: You can divide two integer values using the usual slash (*e1* “/” *e2*) operator. It is an error if a division by zero is attempted.

Modulo
: You can find the remainder of the division of two integer values using the usual percent (*e1* “%” *e2*) operator. It is an error if a modulo by zero is attempted.

Unary Minus
: You can negate an expression using the minus “(−*e*)” operator.

Unary Plus
: You can unary plus an expression using the plus “(+*e*)” operator.

Grouping
: Because parentheses are used to indicate other things in most addressing formats, parentheses can’t be used for grouping. Instead *angle brackets* are used: the expressions look like “< *expr* >” and this may take some getting used to.

Bit-Wise Not
: You can bit-wise AND an integer value using the tilde “(˜*e*)” operator.

Bit-Wise And
: You can bit-wise AND two integer values using the ampersand “(*e1* & *e2*)” operator.

Bit-Wise Or
: You can bit-wise OR two integer values using the vertical bar “(*e1* | *e2*)” operator.

Bit-Wise Exclusive-Or
: You can bit-wise exclusive-or two integer values using the carat “(*e1* ^ *e2*)” operator.

Equal
: You can make equality comparisons, using the “(*e1* = *e2*)” operator.

Not Equal
: You can make inequality comparisons, using the “(*e1* <> *e2*)” operator.

Less Than
: You can make less than comparisons, using the “(*e1* `.lt` *e2*)” operator. Not present in the UCSD native assembler.

Less Than Or Equal
: You can make less than or equal comparisons, using the “(*e1* <= *e2*)” or “(*e1* `.le` *e2*)” operator. Not present in the UCSD native assembler.

Greater Than
: You can make greater than comparisons, using the “(*e1* `.gt` *e2*)” operator. Not present in the UCSD native assembler.

Greater Than Or Equal
: You can make greater than or equal comparisons, using the "(*e1* `>=` *e2*)" or "(*e1* `.ge` *e2*)" operator. Not present in the UCSD native assembler.

Operator Precedence
: The precedence of the various operators is the same as for Pascal.

DIRECTIVES

This section details the pseudo-ops understood by the assembler. In general, these are the same pseudo-ops used by the UCSD native assembler; any exceptions will be noted.

.arch
: The `.arch` pseudo-op can be used to change the microprocessor architecture being assembled. (Not present in the UCSD native assembler.)

```
.arch "name"
```

The *name* of the machine must be a quoted string constant. The names are case **in**sensitive.

p-code-le
: Assemble p-code assembler, little endian. This is the default, if no `.arch` is specified. The default radix will be set to decimal.

p-code-be
: Assemble p-code assembler, big endian. The default radix will be set to decimal.

6502
: Assemble Mos Technologies 6502 assembler (Apple][, KIM-1). The default radix will be set to hexadecimal.

.asciz
: This pseudo-op is similar to the `.ascii` pseudo-op, except that is always emits a NUL (0x00) character after the string.

.else
: See `.if` for documentation.

.end
: Used to denote the physical end of an assembly. All input beyond this point is ignored. It is an error if this directive is not present.

.endc
: See `.if` for documentation.

.endif
: This is a synonym of the `.endc` pseudo-op.

.endm
: See `.macro` for documentation.

.error
: This pseudo-op is used to output a message to the standard error stream. A common use of this directive is to provide a diagnostic announcement of a rejected or erroneous macro call or to alert the user to the existence of an illegal set of conditions specified in a conditional assembly.

The values of all the expressions are concatenated together. If you want spaces between them, use a constant string expression.

```
.print "Oops."
```

This pseudo-op was inspired by the PDP-11 pseudo-op of the same name, but it is not quite identical in operation.

.func
: This pseudo-op identifies a function that returns a value. Two words of space to be used for the function value will be placed on the stack after any parameters. A `.func` is ended by the occurrence of a new `.func`, `.proc`, or `.end`.

```
.func identifier [ , expression ]
```

Where *expression* indicates the number of words of parameters expected by this function. The default is 0.

Symbols defined before the first procedure or function are preserved, however symbols defined within the previous procedure or function are dropped. This gives each function a “clean slate” for its local symbols.

.if

The `.if` pseudo-op is used to conditionally assemble portions of the source code.

```
.if condition
   :
   :
.endc
```

It is also possible to specify an *else* part.

```
.if condition
   :
   :
.else
   :
   :
.endc
```

Conditionals can be nested.

The *condition* must evaluate to either an integer (false is zero, true is any non-zero value), or a boolean value.

.include

This pseudo-op is used to include another source file at this position in the source file.

.macro

The `.macro` pseudo-op is used to define macro-instructions. These can be used to group opcodes together, along with appropriate parameters.

```
.macro pop
pla
sta %1
pla
sta %1+1
.endm
```

It is possible to use a macro anywhere you would use a normal opcode. Parameters are referenced using `%1`, `%2`, *etc*.

.print

This pseudo-op is used to output a message to the standard error stream. A common use of this directive is to provide a diagnostic announcement of a rejected or erroneous macro call or to alert the user to the existence of an illegal set of conditions specified in a conditional assembly.

The values of all the expressions are concatenated together. If you want spaces between them, use a constant string expression.

```
.print "Oops."
```

This is not treated as a fatal error.

This pseudo-op was inspired by the PDP-11 pseudo-op of the same name, but it is not quite identical in operation.

.proc

This pseudo-op identifies a procedure that returns no value. A `.proc` is ended by the occurrence of a new `.proc`, `.func`, or `.end`.

```
.proc identifier [ , expression ]
```

Where *expression* indicates the number of words of parameters expected by this routine. The default is 0.

Symbols defined before the first procedure or function are preserved, however symbols defined within the previous procedure or function are dropped. This gives each procedure a “clean slate” for its local symbols.

.radix

The `.radix` pseudo-op can be used to change the default radix used by the assembler. (Not present in the UCSD native assembler.)

`.radix` *number*

The *number* must be between 2 and 36. You may need to use one of the explicit number forms (next paragraph) to cope with an unknown or undefined default radix.

You can always get a decimal number by using a dot (`.`) suffix. You can always get a hexadecimal number by using a `H` suffix (if the default radix is less than 19), or a C-style `0x` prefix (if the default radix is less than 34).

As with the cross compiler, you can also specify a number with a radix and hash (#) prefix. For example, an octal number could be written `8#377`; other radixes are also possible, such as `13#42`. The radix base before the hash (#) is always decimal, no matter what the default radix has been set to.

Note that the (implicit) `.arch` pseudo-op also sets the default radix.

.sbttl

The `.sbttl` pseudo-op can be used to change the second of two lines of page title of the assembler listing.

`.sbttl` *text*

The text does not need to be quoted, and it may contain spaces. The effect of this opcode will be seen on the next page heading printed. It is not possible to set both the title and the sub-title of the first page.

.title

The `.title` pseudo-op can be used to change the first of two lines of page title of the assembler listing.

`.title` *text*

The text does not need to be quoted, and it may contain spaces. The effect of this opcode will be seen on the next page heading printed. To set the heading of the *first* page, this directive must be the first line in the file.

SEE ALSO

ucsdpsys_compile(1)
A cross compiler from Pascal to UCSD p-System codefiles.

ucsdpsys_disassemble(1)
disassemble a UCSD p-System code file

ucsdpsys_link(1)
UCSD p-System codefile linker

COPYRIGHT

ucsdpsys_assemble version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_charset − UCSD p-System font builder

SYNOPSIS

ucsdpsys_charset −e [*input-text-file* [*output-binary-file*]]
ucsdpsys_charset −V

DESCRIPTION

The *ucsdpsys_charset* program is used to decode and encode font characters for use as the `SYSTEM.CHARSET` file.

OPTIONS

The following options are understood:

−d

−−decode
Decode the binary font file into a text file which can be edited.

−e

−−encode
Encode a text file representation of the glyphs of the font into the binary for used for the `SYSTEM.CHARSET` file.

−i

−−include
Encode a text file representation of the glyphs of the font into C code for an include file to define an array of bytes of data.

−V

−−version
Print the version of the *ucsdpsys_charset* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_charset* command will exit with a status of 1 on any error. The *ucsdpsys_charset* command will only exit with a status of 0 if there are no errors.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_compile − compile Pascal source to UCSD p-System code file

SYNOPSIS

ucsdpsys_compile [*option*...] *filename*
ucsdpsys_compile −V

DESCRIPTION

The *ucsdpsys_compile* program is used to compile Pascal source code to UCSD p-System code files.

OPTIONS

The following options are understood:

−A *name*

−−host=*name*
: This option may be used to specify the machine architecture from the command line, as if there was a `(*$feature host` *name*`*)` comment at the start of the source file. There is an equivalent option for the *ucsdpsys_assemble*(1) command.

−f *feature-name=value*

−−feature=*feature-name=value*
: Set the selected feature to the given boolean value. (List of features may be found below.)

−f *feature-name*

−−feature=*feature-name*
: Set the selected feature to true.

−f no-*feature-name*

−−feature=no-*feature-name*
: Set the selected feature to false.

−I *directory*

−−include=*directory*
: This option is used to specify an include file directory to search. This option may be given more than once.

−J *directory*

−−view-path=*directory*
: This option is used to specify a directory to append to the view path. This option may be given more than once.

−L *directory*

−−library-path=*directory*
: This option may be used to add another directory to the list of directories to be searched for library codefiles. This option may be used more than once, directories will be searched in the order specified.

−−listing=*filename*
: This option maybe used to name a file to accept the compiler listing. By default, no listing is produced. The file name "−" is understood to mean the standard output. The listing will consist of the source code interleaved with the disassembled p-code.

−l

−−long-addresses
: This option may be used to emit addresses in into the listing as "s*n* p*n* o*nnn*" (*i.e.* the segment number, procedure number, and procedure offset). This is the same format used by the *ucsdpsys_cm*(1) **−−trace** option, making it easier to cross reference from the trace back into the

compiler listing. Has no effect without the **--listing** option.

-o *filename*

--output=*filename*
This option may be used to set the name of the output file. It defaults to the name of the source file, with the suffix replaced with "`.code`".

-P *release-name*

--p-machine=*release-name*
This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to "II.1" if not set.

-V

--version
Print the version of the *ucsdpsys_compile* program being executed.

-v

--verbose
Verbose. Statistics about the compilation are printed.

-W *warning-name=value*

--warning=*warning-name=value*
Set the selected warning to the given boolean value. (List of warnings may be found below.)

-W *warning-name*

--warning=*warning-name*
Set the selected warning to true. For example, use "`-Werror`" to turn all warnings into fatal errors.

-W no-*warning-name*

--warning=no-*warning-name*
Set the selected warning to false. For example, use "`-Wno-shadow`" to disable shadow warnings.

-y

--grammar-trace
Turn on parse debugging. Very verbose. Intended for compiler developers only.

All other options will produce a diagnostic error.

ENVIRONMENT VARIABLES

UCSD_PSYSTEM_XC_LIBRARY_PATH
This is a colon-separated list of directories to be searched for codefiles containing library UNIT segments.

EXIT STATUS

The *ucsdpsys_compile* command will exit with a status of 1 on any error. The *ucsdpsys_compile* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_assemble(1)
UCSD p-System cross assembler, for multiple CPU types

ucsdpsys_disassemble(1)
A utility to disassemble UCSD p-System codefiles.

ucsdpsys_link(1)
UCSD p-System codefile linker

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

DIFFERENCES

There are a number of differences between this cross compiler and the UCSD p-System native Pascal compiler.

Policy

In developing the cross compiler, a number of design decisions had to be made. This section is more-or-less how to decide what to do in the interesting places.

- When you can, do static analysis. Finding bugs at compile time is better than finding bugs at run time, or worse, not finding them.
- Evaluate constant expressions at compile time (no warning required). It makes for smaller code, and it also allows more static analysis.
- Adding a new feature is OK, but all new features must issue a warning that the native compiler can't do it, and all new features need a (*$warning*) setting to turn off the warning if the user is doing it on purpose.
- Adding new features for better ISO 7185:1990 Standard Pascal compliance is a good thing.
- If new features are so different that they are disabled by default, and have a (*$feature*) setting to turn them on, they don't need a warning as well.
- There is no need to produce code that is identical to that produced by the native compiler. If you can produce better code, without changing the meaning, or the results, go for it.
- Short circuit boolean evaluations subtly break at last one of the above policies, in the case where function calls within the conditions have side effects. When we have enough flow analysis to know if functions have side effects, we can issue warnings.
- It is essential that parameters are pushed onto the stack identically to the native compiler, or system calls will stop working. The same applies how parameters are allocated space in the stack frame.
- There is more leniency for variable allocations, especially temporary variables.
- There is some evidence, in the system utilities, of using weird negative array indexes to access system internals. There is no need to try for backwards compatibility in this, and thus no need for identical stack frame layouts, in order to preserve stack sizes for said weird negative offsets to continue to "work".
- There is no need to reproduce native compiler bugs, provided that they really are bugs, and not mis-features that programs relied on. This clause is last, because it shouldn't be used very often.

At some point you are going to find stuff that is FUBAR: document it (see below).

Cross Compiler

It is called a cross compiler because, rather than generate native code for the system the compiler is running on, it instead generates code to be run on a different system; in this case the UCSD p-System. The code must be moved across to be executed.

The *ucsdpsys_compile*(1) program does not run natively on a UCSD p-System, so it does not have to be invoked manually. The cross compiler can exist in a modern software development tool chain, and can be executed by build tools such as *make*(1).

You do not have to have a working UCSD p-System in order to compile UCSD Pascal programs. This makes it possible to bootstrap a new UCSD p-System "from scratch".

Error Reporting

The error messages are vastly improved. Wherever possible accurate location data is included in messages, along with names and types.

The location of symbol declarations is tracked, so that errors relating to that symbol can refer back to the declaration location.

If you have spent the last 25 years writing C (or some other language with C-like syntax) the compiler will remind you whenever it sees an equality (=) operator where there should be an assignment (:=) operator. This is more helpful than "syntax error". Some other C-isms are also diagnosed.

Declarations

The values presented to CONST declarations may be any expression, provided it evaluates to a constant value at compile time. A non-constant will be an error (but not a syntax error).

When declaring array types, and integer subrange types, expressions within the declaration may be any expression, provided they evaluate to constant values at compile time.

Boolean Expressions

The cross compiler uses short-circuit evaluation of boolean expressions. For example, if the result of a logical AND expression can be determined from the left-hand side (being false) the right-hand side is never evaluated. Similarly, if the result of a logical OR expression can be determined from the left hand side (being true) the right-hand side is never evaluated.

The cross compiler also makes use of the NFJ and EFJ opcodes that were present in the p-machine used by Apple Pascal, but were never generated by the Apple Pascal native compiler.

The UCSD native compiler did not allow nested SEGMENT procedures and functions. It would emit a "399 Implementation restriction" error if you did so. The cross compiler does not have this restriction.

Lexical Enhancements

It is possible to use hexadecimal notation for integer constants, they have a leading dollar ("$") followed by hexadecimal digits, in either case. This feature is common in modern Pascal implementations, but was not present in UCSD Pascal.

It is possible to use binary notation for integer constants, they have a leading percent ("%") followed by decimal digits. This feature is common in modern Pascal implementations, but was not present in UCSD Pascal.

It is possible to use exponential notation for real constants. That is, numbers such as "1.2e6" are recognized; exponents can be negative, too. This feature is common in modern Pascal implementations, but was not present in UCSD Pascal.

Reserved Words

The following identifiers are reserved keywords in this compiler: OTHERWISE, SCAN, SEPARATE, SIZEOF, WRITE and WRITELN. They were not reserved words in the UCSD native compiler.

In the case of SCAN, SIZEOF, WRITE and WRITELN, this is because these functions do not fall into the regular call grammar. The parameters of these calls are not simple expressions, or have the option of not being simple expressions, but need additional grammar support.

OTHERWISE was added to case statements, see the relevant section of this man page.

SEPARATE is like EXTERNAL but there is no compiler support for this, yet; however this keywords does appear in UCSD p-System sources.

Inline Variable Declarations

It is possible to declare variables in-line with the code.

```
function sqrt(x: real): real;
var
  result, diff: real;
begin
  result := x / 2;
  repeat
    begin
      var approx := result - (sqr(result) - x) / (2 * result);
      diff := abs(approx - result);
      result := approx
    end
  until
    diff < 1e-6;
  sqrt := result
end
```

The *approx* temporary variable is created on-the-fly and the symbol is forgotten at the end of the enclosing `begin end` scope. The storage it used is released, and may be re-use in another, later, `begin end` scope. The `repeat until` statement is slightly different, variables declared in-line persist until the end of the until condition, so that variables create inside the loop may be used to control the loop.

This feature was not present in UCSD Pascal. It was introduced to the cross compiler principally to verify the temporary variable behavior required by some `with` and `for` statements.

Bit-wise Operators

The compiler has some extensions present in modern Pascal compilers, allowing bit-wise operations on integers,

In addition to logical NOT expressions, the cross complier understands bit-wise NOT expressions. They have the same notation and precedence, but take an integer operand and the result is an integer.

In addition to logical AND expressions, the cross complier understands bit-wise AND expressions. They have the same notation and precedence, but take integer operands and the result is an integer.

In addition to logical OR expressions, the cross complier understands bit-wise OR expressions. They have the same notation and precedence, but take integer operands and the result is an integer.

The bit-wise operators do not use short-circuit evaluation.

Labels and Goto

It is possible to have named labels, not just numeric labels. Warnings rather than errors are issued when a `goto` statement is used.

If Statement

As mentioned in Boolean Expressions, above, the if statement uses short circuit evaluation for the control statement.

If control expression is constant, only the code for the relevant branch is generated.

Case Statement

It is possible to place an OTHERWISE clause at the end of CASE statements. It will be used for all values not matching one of the preceding case values.

Case values (to the left of the colon) may be any expression, provided they evaluate to constant values at compile time.

Pathological uses of CASE statements that produce a huge code explosion are diagnosed, and a much more informative error message is produced than that of the UCSD p-System native compiler.

The native compiler generates XJP opcodes with the following table always pointing backwards to each of the cases (each is positive). This requires an unconditional branch around all of the cases to the XJP opcode at the end. The `(*$feature short-case true*)` control comment may be used to reverse this order. This generates code that is slightly smaller, slightly faster, and uses forward pointing self relative pointers (*i.e.* each is negative). While negative self-relative pointers work correctly on the Klebsch implementation, it isn't known if they work correctly on all p-machine implementations (they should, signed and unsigned subtraction are the same thing on twos-compliment machines).

Code Size

The native UCSD p-System compiler was constrained in the amount of memory it had available for generating code. Function bodies were limited to about 1200 bytes of code, and segments were limited to about 28000 bytes total for all functions in the segment.

The *ucsdpsys_compile*(1) command is able to generate segments as large as 65534 bytes, which is patently overkill for running on a 64kB system, because you wouldn't be able to load it into memory, the system segments wouldn't have left enough room. Functions can be as large as you want, provided they all fit into their segment. Truly huge procedures may run out of jump table entries, however, because there is no way to increase the limit of 64.

The native UCSD Pascal compiler also has a limit of 140 functions per segment. This, again, was a memory size constraint. The cross compiler can have up to 255 functions per segment (the limit of

addressability) without difficulty.

When possible constant expressions, or constant parts of expressions, are evaluated at compile time, and inserted into the code as constants. This is usually less code than the native compiler produces.

Control Comments

There are a number of control comments that have been added, to fine tune the operation of the cross compiler. All control comment names are case-**in**sensitive.

(*$b*) Synonym for (*$feature big-endian true*)

(*$d+*) Synonym for (*$feature debug true*)

(*$f*) Synonym for (*$feature big-endian true*)

(*$feature *name value*)

This control comment is used to set a number of compiler features. You can also control features from the command line, using the **−f** option. The features are:

big-endian [*bool*]

This control comment tells the compiler will produce big-endian p-code rather than the default little-endian. Defaults to false.

chr-range-check *bool*

This control comment is used to enable (true) or disable (false) the generation of range checking opcode around CHR parameters. Defaults to false.

The range checks are only issued if this feature and the regular *range-check* feature are both true.

debug [*bool*]

This control comment many be used to turn on (true) or off (false) the generation of break point (source code line) opcodes into the output. Defaults to false.

efj-nfj *bool*

This option may be used to control the use of EFJ and NFJ opcodes (used to optimize some branch conditions) in the unlikely case where they are not available on your p-Machine implementation. (The Apple p-machine has them, and so does the Klebsch p-machine). Defaults to true, generate these opcodes.

extra-set-comparisons *bool*

This option says to the LES POWR and GRT POWR opcodes. These were not present in the original UCSD p-machine, and the compiler does not use them by default.

ignore-undefined-segment-zero *bool*

The compiler always checks for procedures and functions that were declared forward, but were not later defined, and issues a fatal error for each such symbol. This option tells the compiler to ignore procedures from segment zero that were declared forward, but were not defined. This is only of use to system utilities. Defaults to false.

You can turn this on and off for specific symbols. The setting takes effect for all subsequent function and procedure declarations, if they are in segment zero. To affect all of them, you must put the control comment at the start of the file, or on the command line.

inline-math *bool*

This flag is used to enable or disable the use of built-in math and trig functions that correspond to p-machine opcodes. These were not in the Apple Pascal p-Machine (presumably) due to size constraints, but the p-machine spec still defines them.

This flag defaults to false, meaning you have to use the TRANSCENDENTAL unit if you want math functions.

This flag must be set prior to the PROGRAM or UNIT keywords, as it affects the contents of the built-in symbol table.

iocheck *bool*
: This control comment tells the compiler whether or not to issue IOCHECK opcodes after I/O statements. Defaults to true.

little-endian [*bool*]
: This is the opposite of the *big-endian* option. Defaults to true.

long-integer-constants *bool*
: This option may be used to control constant folding of INTEGER[*n*] expressions, and the presence of long integer constants in the code. Defaults to true.

long-integer-extensions *bool*
: This option may be used to control the use of INTEGER[*n*] opcode extensions (ABS, MOD, ODD, SQR) in the p-machine. Defaults to false (most p-machine implementations don't have them).

maximum-name-length *integer*
: The maximum length of an identifier. The setting must be in the range 8..32767. Defaults to 8, just as the UCSD native Pascal compiler did.

range-check *bool*
: This control comment is used to enable (true) or disable (false) the generation of range checking opcode around array indexing and some assignments. Defaults to true.

short-case *bool*
: This control comment may be used to enable (true) or disable (false) the use of a shorter technique to generate CASE statements. Defaults to false.

short-with *bool*
: This control comment may be used to enable (true) or disable (false) the use of a shorter technique to generate the implicit dot expressions required by WITH statements. When enabled, if the base address in the WITH statement is simple enough, not temporary pointer value is created. When disabled, or when the base address expression is sufficiently complicated, a temporary pointer variable is used, just as in the UCSD native compiler. Defaults to true.

tiny *bool*
: This control comment may be used to enable (true) or disable (false) several of the built-in functions. This was a space-saving measure in the UCSD native compiler. Defaults to false.

underscore-significant *bool*
: This control comment may be used to modify the significance or underscores in identifiers (names). Setting to true gives ISO 10206 conforming behavior. Defaults to false, just as the UCSD native compiler did.

user *bool*
: This control comment may be used to enable (true) or disable (false) user-mode compiling. Defaults to true.

  The use of (*$feature user false*) produces system programs which use a different set of segments, and disable a number of other checks. This is used when compiling the UCSD p-System itself, and a number of other system utilities.

Other feature names will elicit diagnostic error messages.

(*$g+*) Synonym for (*$warning goto false*)

(*$I−*) Synonym for (*$feature io-check false*)

(*$I *filename* *)
: Synonym for (*$include *filename**)

(*$include *filename* *)
: Include the named source file at this point in the code. The filename may not contain white space or comma characters.

(*$r−*) Synonym for (*$feature range-check false*)

(*$r+*) Synonym for (*$feature range-check true*)

(*$t+*) Synonym for (*$feature tiny true*)

(*$u−*) Synonym for (*$feature user false*)

(*$warning *name value* *)
: This control comment is used to enable and disable the various warnings produced by the compiler. You can also control warnings from the command line, using the **−W** option. The warnings are:

  address-of *bool*
  : This enables (true) or disables (false) the warning the accompanies the use of the address-of operator. The address-of operator "@" allow you to take the address of a variable. Most modern Pascal implementation have this, but the original UCSD p-System Pascal did not. Defaults to true.

  constant-branch *bool*
  : This enables (true) or disables (false) warnings about constant control expressions for IF, WHILE, REPEAT UNTIL, and CASE statements. Defaults to true.

  binary-constant *bool*
  : The ability to write binary constants is a common feature of modern Pascal implementations, however they were not present in UCSD Pascal. When this flag is true, warnings are issued for binary constants (%01010) in the source code. When this flag is false, binary constants are silently accepted. Defaults to true.

  empty-parentheses *bool*
  : This control comment may be used to enable (true) or disable (false) the warning that accompanies the use of empty parentheses for function calls and declarations. This is a C coder coping strategy. Defaults to true.

  error [*bool*]
  : When this flag is true, all enabled warnings are treated as compile errors. When this flag is false, warnings do not cause the compile to fail. Defaults to false.

  goto *bool*
  : This control comment may be used to enable (true) or disable (false) the warning is issued when the GOTO statement appears in the source code. Given that goto is considered harmful, it defaults to true.

  hex-constant *bool*
  : The ability to write hexadecimal constants is a common feature of modern Pascal implementations, however they were not present in UCSD Pascal. When this flag is true, warnings are issued for hexadecimal constants ($XX) in the source code. When this flag is false, hexadecimal constants are silently accepted. Defaults to true.

  named-label *bool*
  : Named labels are a common feature in modern Pascal compilers, but they are not present in the native compiler. When this flag is true, named labels are complained about. When this flag is false, named labels are silently accepted. Defaults to true.

  otherwise *bool*
  : This option controls whether or not to issue a warning when an OTHERWISE clause is seen attached to a CASE statement. This is common in modern Pascal implementations, but was not present in UCSD Pascal. Defaults to true.

shadow *bool*
: Shadowing occurs when declaring a symbol (function, variable, *etc*) blocks access to a symbol declared earlier. This almost always creates a maintenance problem. Set this to true to issue warnings when symbols are shadowed, set this to false to silence shadow warnings. Defaults to true.

set-comparisons *bool*
: This warning is issued if set<set or set>set comparisons are mode. The UCSD native Pascal compiler did not accept such expressions.

silent *bool*
: When this flag is set to true, no warnings at all are produced (this is effectively a warning master disable). When this flag is set to false, warnings are printed. Defaults to false.

Other warning names will elicit diagnostic error messages.

ternary-expression *bool*
: This option is used to enable or disable warning when C-style ternary expressions (e1?e2:e3) are encountered. Defaults to true.

unreachable *bool*
: This option is used to enable or disable warning when unreachable statements are found by the compiler. Unreachable statement are those for which the follow of execution does no flow into them from the previous statement, and statements that do not have a label. By default, this warning is enabled.

Compiling Separate Units

The UCSD native compiler is able to compile programs in several separate compilation units, an either link them with the system linker, or link them implicitly at runtime if they are intrinsic units. How this was accomplished evolved over time, the cross compiler duplicates the II.1 implementation.

Some examples may help. This following code is what you may expect of a separate unit

```
(*$S+*)
separate unit frog;
interface

  const fly_size = 10;
  type wart_type = (green, brown);
  procedure jump(dist: integer);
  function warts: integer;

implementation

  const pi = 3.14159;
  type etc = 0..13;

  procedure jump;
  begin
  end;

  function warts;
  begin
    warts := 0;
  end;

end.
```

The II.1 compiler issues an error if the (*$S+*) was not present, the cross compiler will issue a warning. The II.1 compiler issued an error for the SEPARATE keyword, the cross compiler issues a warning.

System units, ones that needed to access the system global variables at lex level −1, used a different syntax. This was never documented anywhere, as far as I can tell.

```
(*$S+*)
(*$U−*)
program pascalsystem;

  separate unit frog;
  interface

    const fly_size = 10;
    type wart_type = (green, brown);
    procedure jump(dist: integer);
    function warts: integer;

  implementation

    const pi = 3.14159;
    type etc = 0..13;

    procedure jump;
    begin
    end;

    function warts;
    begin
      warts := 0;
    end;

  end;

begin
end.
```

The II.1 compiler issues an error if the (*$S+*) or (*$U−*) was not present, the cross compiler will issue a warning. The II.1 compiler issued an error for the SEPARATE keyword, the cross compiler issues a warning.

In summary: omit the SEPARATE keyword.

FUBAR: Fouled Up Beyond All Recovery

See the “Policy” section, above.

The CHR built-in doesn’t actually do anything. That means that chr(32767) has the same value on the stack, it doesn’t do a C-style value cast of masking its operand with 0xFF. Most of this brain damage can be found by using the the (*$feature chr-range-check true*) control comment. Constant foul ups will be found at compile time.

The ODD built-in doesn’t actually do anything, the same vales is on the stack, it doesn’t do a C-style cast of masking its operand with 1. The boolean branch opcodes just look at the bottom bit. Constant folding by the compiler doesn’t have this behavior.

NAME

ucsdpsys_depends − UCSD Pascal file dependency tracker

SYNOPSIS

ucsdpsys_depends [*option*...] *filename*...
ucsdpsys_depends −V The *ucsdpsys_depends* program is used to scan a Pascal source file for include directives.

OPTIONS

The following options are understood:

−I *directory*

−−include=*directory*
This option is used to specify an include file directory to search. This option may be given more than once.

−J *directory*

−−view-path=*directory*
This option is used to specify a directory to append to the view path. This option may be given more than once.

−L

−−one-per-line
The dependencies are usually written all on the one line. This option requests that each be on a separate line.

−o *filename*

−−output=*filename*
This option may be used to specify the output file. Defaults to the standard output.

−P *string*

−−prefix=*string*
The prefix to put at the start of the output line.

−r

−−recursive
This option may be used to request recursive analysis. Because *cook*(1) cascade directives are being used, recursive analysis is rarely needed.

−S *string*

−−suffix=*string*
The suffix to put at the end of the output line.

−V

−−version
Print the version of the *ucsdpsys_depends* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_depends* command will exit with a status of 1 on any error. The *ucsdpsys_depends* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_compile(1)
UCSD p-System Pascal compiler

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_disassemble − disassemble a UCSD p-System code file

SYNOPSIS

ucsdpsys_disassemble [*option...*] *filename*
ucsdpsys_disassemble −V

DESCRIPTION

The *ucsdpsys_disassemble* program is used to

OPTIONS

The following options are understood:

−a

−−no−address
Do not include addresses in the output. This makes automated testing easier when just a byte or two is added, it stops the diff being unhelpfully large.

−c

−−comment
Add a descriptive comment for each opcode.

−e

−−extend
This option is used to include the structure of the codefile itself in the listing.

−o *filename*

−−output=*filename*
Write the output to the named file, rather than the standard output.

−P *release-name*

−−p−machine=*release-name*
This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to “II.1” if not set.

−V

−−version
Print the version of the *ucsdpsys_disassemble* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_disassemble* command will exit with a status of 1 on any error. The *ucsdpsys_disassemble* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_assemble(1)
UCSD p-System cross assembler, for multiple CPU types

ucsdpsys_compile(1)
A cross compiler from Pascal to UCSD p-System codefiles.

COPYRIGHT

ucsdpsys_disassemble version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_downcase − convert Pascal to lower case

SYNOPSIS

ucsdpsys_downcase *filename*...

ucsdpsys_downcase −−version

DESCRIPTION

The *ucsdpsys_downcase* program is used to convert Pascal source code from upper case to lower case. It leaves the contents of string constants, character constants and comments unaltered, but converts all program text to lower case.

Pascal is case-**in**sensitive for all identifiers, so this is a safe thing to do. It will not change program semantics.

Each file named on the command line will be converted in place. If no files are named, the standard input is converted and written to the standard output.

OPTIONS

The following options are understood:

−V

−−version

Print the version of the *ucsdpsys_downcase* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_downcase* command will exit with a status of 1 on any error. The *ucsdpsys_downcase* command will only exit with a status of 0 if there are no errors.

COPYRIGHT

ucsdpsys_downcase version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_errors − UCSD p-System assembler error file builder

SYNOPSIS

ucsdpsys_errors [*option...*][*infile* [*outfile*]]]]
ucsdpsys_errors −−version

DESCRIPTION

The *ucsdpsys_errors* program is used to translate an assembler error file from its text for to its binary form, and back again.

OPTIONS

The following options are understood:

−A *name*

−−architecture=*name*
This option is used to specify the architecture of the assembler in use.

−d

−−decode
This option is used to specify a translation from binary to text form is required. This can be useful for reverse engineering the text when all you have is the binary form.

−e

−−encode
This option is used to specify a translation from text to binary is required. The binary form is used to simplify the error reporting code.

−V

−−version
Print the version of the *ucsdpsys_errors* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_errors* command will exit with a status of 1 on any error. The *ucsdpsys_errors* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_errors(5)
UCSD p-System assembler error file format

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_history − UCSD Pascal notes and archaeology

DESCRIPTION

This document is an attempt to collect together historical information about the UCSD Pascal system. Including who wrote it, notes about is various idiosyncrasies, and how to boot a p-System “from scratch”.

It may be that this information more properly belongs in Wikipedia. [http://en.wikipedia.org/wiki/Ucsd_pascal] With an eye towards this, all sources are being annotated with their URL.

ARCHAEOLOGY

This section covers the historical period from 1977-ish, when UCSD Pascal project was initiated, to 1983-ish, when progress of languages and systems had basically left UCSD Pascal behind.

Predecessors

Urs Ammann, a student of Niklaus Wirth, originally presented a p-code in his PhD thesis − see Urs Ammann, *On Code Generation in a Pascal Compiler*, Software Practice and Experience, Vol. 7, No. 3, 1977, pp. 391-423, from which the UCSD implementation was derived, the Zurich Pascal-P implementation. The UCSD implementation changed the Zurich implementation to be “byte oriented”. [15] [16]

Using a tool developed in response to the SCO kerfuffle (http://en.wikipedia.org/wiki/SCO−Linux_controversies), Andrew Tridgel’s *hashmatch* tool (http://samba.org/~tridge/hashmatch/), or even the venerable *diff*(1) command, it can be seen that there are significant similarities between the P2 compiler (http://www.standardpascal.com/p2/pcomp.pas) and the UCSD Pascal I.3 compiler posted to CompuServe (http://tech.groups.yahoo.com/group/UCSDPascal/files/Compiler/). The *numbers* and the names of the opcodes for both the P2 compiler and the UCSD 1.3 compiler are all but identical.

The P2 compiler was made available in 1974, and Ken Bowles obtained a copy that same year, and saw that it had the potential to be ported to the microprocessors of the day [1]. The project that was to be UCSD Pascal was up and running by Oct−1974 [2].

PUG Newsletter 4

[9] 1975-Aug-22. We are indeed working with PASCAL on the B6700. Whether the work is of immediate interest to you is another question. Making PASCAL into a stable B6700 product for users is a secondary objective of our project. Our primary aim is to create an interactive student debugging environment on the PDP-11 with virtually all of the software written in PASCAL.

[9] The overall objectives of the project are described in the enclosed project prospectus. Students will interact with PASCAL on the small machines in a manner very reminiscent of APL on IBM 360/370. PASCAL is interpreted using a modified version of the Zurich P-Machine recently released. The main purpose of the modifications is to reduce the the size of the compiled code so that the PASCAL compiler can fit within core of a small the limited machine. Yes, we have done the same kinds of statistical studies represented in the reports you kindly sent, though our data is not in as elegant a form. We are confident that the compiler can be run on a PDP-11 with at least 20K words of memory. We are hoping to reduce that amount further to perhaps 16K, when time permits. Currently the interpreter is operating, but has yet to be tried with the whole compiler on the PDP-11.

[9] We are using the modified PASCAL compiler on the B6700 as a tool for developing the new PASCAL system, and generating pseudo-code for the PDP-11. Having started with an interpreter for the Zurich P-Machine, we have progressed through various stages of bootstrapping to get a system compatible with the PDP-11 objective, and the interactive system objective. Concurrent with the work using the interpreter, we also have an advanced student programmer writing an assembler which converts the compiler P-code output into directly executable B6700 code. The B6700 code has been executed with small programs, and should be running the whole expanded a week or so. This compile-assemble system manages its memory in one large array in a fashion similar to that used on conventional machines. We are using the B6700 SWAPPER for much of our batch work, and hence have been able to use DIRECT (non-overlayable) array

space for this purpose to enhance the speed of the processing.

[9] The short-term objective for the B6700 compile-assemble system is to provide a back-up means for students to use for PASCAL homework problem starting in late September. Our PDP-11 equipment is not all here yet, and we clearly will not be ready to use the small machines with students during the first few weeks of the Fall Quarter. Over the time period of the academic year about to start, we will almost certainly have someone complete the job of making a PASCAL compiler that can generate B6700 code directly. Yet to be resolved is the question of whether we can map the PASCAL data structures into the array-row structure of the B6700 without doing violence to the basic approach of the P-compiler.

[9] The interpretive system is slow on the B6700, as might be expected. The major consumer of time is the low level character processing in the INSYMBOL and NEXTCH procedures. We have changed these procedures completely, so as to depend upon installation intrinsic functions (Standard Procedures) that make use of the B6700 string processing hardware. The GETSYM intrinsic returns information on each successive token in an area of stack that serves as a scanner information block. This provides a clean interface between compiler and interpreter, but it runs about half as fast as an earlier less-clean version (part way through the bootstrapping) in which virtually all of the work of INSYMBOL was done in an ALGOL intrinsic. The current B6700 interpretive version takes about 10 minutes of processor time to compile the source file from Zurich. We expect the compile-assemble version, and also the PDP-11 version, to run roughly five times faster than that.

[9] During the next two months we will be up to our ears in alligators getting this system completed well enough to use for teaching. At a later stage I would be happy to share more details with you.

PUG Newsletter 8

[10] 1977-Apr-17. UCSD has recently started using a single user software system for microcomputers, with all major programs written in PASCAL. The compiler is based on the P-2 portable compiler distributes by the ETH group at Zurich, but it generates compressed pseudo-code for a much revised P-machine interpreter. As currently implemented on the LSI-11, compile speed is about 700 lines per minute (1000 on the PDP-11/10). The system includes an interactive monitor, editor, utility file handler, and debugging package in addition to the compiler and interpreter. With 56K bytes of main memory, and dual floppy disk drives, it has proven more convenient and faster to do all software development on the microcomputer than to cross compile from a big machine. Whereas we have been using version of this system that depend on I/O support from the RT11 operating system distributed by Digital Equipment Corp., our new system is independent of any external software support. The resident monitor, in interpreter, and run-time support package occupy an aggregate of about 10K bytes of memory.

[10] Operation of large programs is facilitated through the concept of “Segment Procedures”, which are rolled into memory only while actually invoked. The compiler (20K bytes), editor, and File handler are all separate segment procedures. One segment procedure can call others, and segment procedure may be declared nested within other segment procedures, to allow flexibility in memory management. The user’s data space expands (or contracts if necessary) to take advantage of as much memory as possible after the appropriate code segment have been loaded.

[10] Our plan is to have the new system completed to the point where it may be released to others by mid summer, 1977, with documentation package included. During the summer, we also plan to complete a graphics support package (including an editor for graphics oriented CAI materials), an assembler for PDP-11 native code, and a compiler option allowing selected PASCAL procedures to generate native code rather than P-machine pseudo code. The system is designed to make relatively painless the problem of adding native code routines programmed in assembly language, allowing a user to augment the set of built-in functions and procedures where efficiency is important. This note has been composed and printed using a proprietary extended version of the text editor intended for use with a CRT display, which should be ready for release by late summer. The system should be usable on any PDP-11 system capable of bootstrap loading from RX11-compatible floppy disk drives, or from the drives supplied with the Terak Corporation LSI-11 based machines (see next section). Further details may be obtained, on request to the address given in the heading, in separate notes titled “Status of UCSD PASCAL Project”, and “Preliminary Description of UCSD PASCAL Software System”.

[10] In addition to the well advertised PDP-11/03 systems available from Digital Equipment, several

smaller companies are offering stand-alone computers based on the LSI-11 that would be directly suitable for our software. We have been particularly interested in using a stand-alone machine with low cost graphic display for interactive educational applications. In connection with the EDUCOM Discount Program (see EDUCOM Bulletin, Spring, 1977), it now appears virtually certain that the Terak Corporation 8510A will be available to member institutions for about $5300 per machine (LSI-11, 56K byte RAM, single floppy disk, CRT for superimposed but independent text and graphics, keyboard, RS232 asynchronous interface for network or printer connection).

[10] Anyone who attended the West Coast Computer Faire in San Francisco should have come away impressed that small stand-alone microcomputers are big business and here to stay. It is possible to re-implement our PASCAL based software system on system based on any of the most popular microprocessors within about 3 months of work by one programmer. At UCSD we have started to re-implement for the Zilog Z80 OEM series of modules, which could serve as the basis for PASCAL interpretive operation roughly as fast as the LSI-11. At the Faire, we talked with principal officers of most of the well known microcomputer manufacturers who sell to the hobbyist market, and encountered almost uniform enthusiasm for the idea of making PASCAL available on an industry-wide basis. On the basis of those conversations, there is a reasonable chance that our PASCAL system will be available later this year for use with the 8080A, 6502, and M6800 microprocessors in addition to the LSI-11 and Z80.

[10] There is widespread frustration, among those who make and sell microcomputer systems, that only BASIC is generally available, and that no two BASIC implementations are alike. Many of those we talked with at the Faire asked whether PASCAL could be standardized, to avoid the problem they encounter w1th non-standard BASIC (in addition to providing a more powerful programming vehicle). Even a casual reading of the PASCAL User Group newsletter is enough to convince one that: a) people are finding it necessary to enhance PASCAL for their own particular applications; b) the heterogeneity of the enhancements already reported is so great that no committee exercise is likely to produce a standard.

[10] As an alternative, we believe that a chance exists to establish1sh a defacto standard for PASCAL, at least for small systems, by starting a bandwagon effect in the microcomputer industry. A good definition of the underlying language for such a standard is contained in the Jensen and Wirth “PASCAL User manual and Report”. To implement a complete interactive software system, with adequate efficiency to run on a microcomputer, we have found it necessary to add built-in functions and procedures for handling text and graphics, and an EXIT(*procedure-name*) built-in for clean termination of highly recursive programs. We have implemented SETs of up to 255 members in a way that uses memory efficiently, as well as packed arrays of BOOLEAN, for READ from a keyboard, the implied GET has to happen before the implied transfer from the window variable associated with the file. For handling floppy disks and other small storage media, we use the DEC standard of 512 byte blocks, and allow logical records conforming to any structured type allowed in PASCAL. In most other respects we have been able to conform closely to the language defined in the Jensen and Wirth book.

[10] If one common PASCAL based software system were to be come available almost simultaneously for most of the mass distributes microcomputers that system would establish the basis of a defacto standard for small stand-alone computers. Changes to such a system would certainly be needed with experience, but those changes might well be made readily available to most user through “down line loading” of object code through the dialed telephone network. Control of the PASCAL language standard might well be vested, at least temporarily, in a committee appointed by the PASCAL User’s Group. Fast turnaround communications among members of such a committee could be supported by “electronic mail” techniques over the dialed telephone network. The verbal responses we received from the manufacturers at the Computer Faire suggest that an unusual opportunity, that may not be repeated, exists in mid 1977 to establish a defacto standard in the manner described here. We invite the PASCAL User’s Group to join with us at UCSD in bringing this about this summer. In most respects, the language and system definition design questions can be separated from implementation details. We have sought support to allow some of the advanced computer science students at UCSD to perform the implementation work on as many of the microcomputers as possible. Representatives of other institutions would be welcome to work with us in La Jola, either on’ system definition or implementation. However we will not be able, ourselves, to devote a major percentage of out working time on definition of a standard.

[10] Introductory Textbook. For the last two years we have used PASCAL as the basis of the large attendance introductory course in problem solving and programming at UCSD. The course is based on a textbook by this writer, that so far has been printed in the campus print shop. Student responses have been unusually favorable, and the course reaches more than two-thirds of the undergraduate population even though it is treated as elective for most majors. This response results partly from the non-numerical approach of the book, partly from student interest in our interactive system on the PDP-11s, and partly from our use of Keller's Personalized System of Instruction (PSI) as the teaching method. Though suitable for PSI, the book can also be used as the basis for a conventional course. At the invitation of Professor David Gries, acting as computer science areas editor for Springer Verlag publishers, the book will be published in paperback form this summer. The production schedule will be tights, and we anticipate that the first copies will be available barely in time for the start of fall quarter classes in late September. Springer is interested in knowing who might be interested in using the book and when. Unfortunately, alterations to make the non-numerical approach more readily accessible on many machines will make it difficult to circulate advance copies of the final text until late June at the earliest. We will be happy to forward inquiries to Springer.

[10] Though very popular with the students, the non-numerical approach of the book has been difficult to sell to most other publishers. The approach used in fact has depended upon programming examples using English text, and requires STRING variables and supporting built-in functions that we have added to PASCAL. In spite of this, the students learn the same programming skills that are taught in courses using traditional algebraic problem examples.

[10] Since the inception of our project, have wanted to orient the courses to teaching with graphic oriented problem examples, using an approach motivated by the "Turtle Graphics" used by Seymour Papert of MIT. The microcomputers now becoming available make it possible to teach with a graphics orientation virtually no higher price than needed for non-graphic materials. Accordingly, the textbook will be revised to augment and often replace the text oriented examples with graphics examples. For potential users lacking a microcomputer with graphics display, several alternate possibilities exist. Out built-in functions and procedures for graphics should be relatively easy to add to existing PASCAL compilers for other machines, and we will supply documentation to assist in that process. A description of the built-in's needed is contained in the note "Status of UCSD Pascal Project" already cited. The implementation will assume a graphic display based on the "bit-map" principle, for which many devices are available in the microcomputer industry. Alternate display drivers will also be provided for the Tektronix 4006, 4010, ... series of direct-view storage tube terminals. Successful, though crude, plotting of the graphic output will also be possible on ordinary line printers. High quality graphic output is possible on matrix printers such as those made by Printronix Gould, Varian, and Versatec.

[10] B6700 PASCAL Compiler. A PASCAL compiler which generates native code for the B7600 is now in operation at UCSD and available for distribution from the UCSD Computer Center. The compiler is written in PASCAL, and is based on the same variant of the P-2 portable compiler on which we have based the microcomputer implementation. Compile speed is about 5000 lines per minute of logged processor time. This compiler has been used for teaching large classes at UCSD for th e last two months. As far as we know, most of the serious bugs in the original P-2 compiler have been corrected in both the B6700 and microcomputer implementations. The B6700 compiler provides access to most of the extensive file handling features of the B6700. At present, no implementation documentation has been completed for the B6700 compiler. The Computer Center will almost certainly generate such document given an indication of interest in using this compiler by other institutions.

[10] Apology to Correspondents. I offer an apology to the many people interested in our PASCAL work who have tried unsuccessfully to reach me by telephone or letter in the last few months. Currently I must depend upon several pooled secretaries who are not easily accessible. Having been occupied with a heavy teaching schedule, and with a committee assignment consuming one or two full full working days per week, the correspondence has piled up. The series of titled notes and position papers cite earlier have been generated in self defense as a way to answer the many enquiries. The committee assignment has entered a down period. Future written requests for these papers will be answered promptly, but telephone inquiries many remain difficult until the re-write of the book is completed.

[10] p. 63: Zilog Z-80. Ken Bowles has announced an implementation for the Z-80 to be distributed sometime this summer. For more details see the Digital Equipment PDP-11 section of this Newsletter.

[10] p. 63: According to Jim C. Warren, editor or Dr. Dobbs Journal of Computer Calisthenics and Orthodontia (Oct. 1976 issue, p. 6), Neil Colvin of Technical Design Labs, Trenton, New Jersey, has adapted a p-code compiler for the Z-80. The p-code interpreter reportedly occupies about 1K bytes. Another Zilog rumor is that Dean Brown is the person at Zilog to see about Pascal.

PUG Newsletter 9, 10

[11] p. 112: Zilog Z-80. Ken Bowles and co-workers, UCSD, have adapted the San Diego DEC LSI-11 implementation to run on the Z110g Z-80 running (at 2.5 MHz) about 70% as fast as the LSI-11. Release is expected by the end of 1977. See the DEC LSI-11 (San Diego) note, above.

PUG Newsletter 11 (1978-Feb)

[12] Status of UCSD PASCAL Project (27 November, 1977)

[12] This is a brief report on the current status of the UCSD PASCAL project intended to answer the questions of the hundreds of people who have been writing to us or calling by telephone. It is our intention eventually to reach a steady state in which we can afford to have full time help capable of responding to such inquiries. For the present, we have to apologize once again to those who may have been kept too long waiting for replies.

[12] 1. Nature of the Project

[12] The project is one of the principal activities of the Institute for Information Systems (IIS). Like other “Organized Research Units”, IIS is operated primarily to provide resources and activities within which students and faculty can conduct research and development projects. Within the range of such activities, projects may support instruction and other public services, though the more usual activities of an ORU involve only basic research.

[12] Under IIS we have developed a major software system for stand-alone microcomputers based on the PASCAL language. The initial reason for developing the system was to support instruction activities at UCSD. However, the system is designed for general purpose use, particularly for the development of interactive software, and for software development in general. The system has matured sufficiently that we are distributing copies to outside users at a $200 fee which pays for some of the student part-time assistants who provide support to users and maintain the software. Under prevailing University policies, we are not attempting to recover capital costs from the fees paid by users of our software package. However, a number of interested industrial firms have provided assistance to further the project through unrestricted grants to the Regents of the University of California marked for use by our project. These grants are our principal source of operating funds at the present time.

[12] Since the PASCAL based software system was developed with the intent to support long term instructional projects, we have placed very high emphasis on machine independence. We expect the repertoire of instructional software developed to use the underlying system to grow very large. The development costs for the instructional software will eventually dwarf the costs of the hardware on which it operates. Since the industry is introducing new microcomputer designs at a rapid rate, we wanted to be able to move the entire software repertoire to new machines with a minimum of effort. As will be detailed in later sections of this note, our system is now running on 5 dis-similar processors, with more planned in the relatively near future. We are using the Digital Equipment LSI-11 for teaching. Versions for the 8080 and Z80 microprocessors are operational and will be ready to distribute on or about 1 January, 1978.

[12] We intend to continue promoting the use of our PASCAL-based system on as many popular microprocessors as practical for two reasons. First, this should provide IIS with a source of continuing income to pay for student projects. Second, PASCAL with extensions is a superior language for system programming, and we believe that it is in the public interest to assist in the current effort of many people and institutions to promote wider use of PASCAL in place of some of the earlier high level languages. Though PASCAL may have some shortcomings for specific applications, when compared to specific proprietary languages, we regard it as by far the best general purpose language now in the public domain.

[12] Our current Research and Development interests include:

[12] a. Methods of making large software systems like ours more readily transportable to new processor architectures.

[12] b. The use of microcomputers as intelligent communications devices to assist humans to work together even when located thousands of miles apart. This interest will eventually involve us in a variety of complex software issues. In the near term it will provide us with an efficient method of supporting users of our software system who are remote from UCSD.

[12] c. Joint use of microcomputers and Keller's Personalized System of Instruction (PSI) as a means of offering high quality college level mass education at lower costs per enrolled student than associated with conventional methods. A published introductory textbook on problem solving using PASCAL, and a library of automated quizzes and record keeping software to go with the textbook, are available to others as a first step in this direction.

[12] d. Exploration of possibilities and software problems associated with new hardware devices or architectures- adaptable to the purposes already described. Examples include video disks, low cost X-Y input devices, and low cost strategies for interconnecting many semi-independent microcomputers.

[12] Partly as a matter of self preservation, we have become interested in. the problem of standards for the PASCAL language. The United States Defense Department and many large industrial corporations have recently decided to use PASCAL as a base language which they would extend, and possibly alter, to create system implementation languages. Although almost every organization has chosen to extend or alter in slightly different ways, we have found that the intent portrayed in most instances is very similar. In our own implementation, we too found it necessary to extend PASCAL, and in very minor ways to alter the base language as described in Niklaus Wirth's widely read "Report" on the language (see .Jensen, K. and N. Wirth, "PASCAL User Manual and Report", Springer Verlag, 1975). We, and many others in the PASCAL User Group, are very much concerned that all this extension and alteration activity will result in PASCAL going the way of BASIC for which hundreds of dialects are in common use. We believe that a chance still exists to gain consensus on a substantial family of PASCAL extensions for system programming, provided that this can be brought about within the next 6 to 12 months. Unless someone does so before us, we intend to convene a summer workshop for representatives of some of the major using organizations in the hope that such a consensus can be reached.

[12] Another ancillary activity of the project has been continuing search for low cost microcomputer hardware of high quality for use in educational institutions − particularly ours. We have been advising and collaborating with EDUCOM regarding establishment of quantity purchase discounts for stand-alone microcomputers. The first microcomputer to be included in the EDUCOM discount program is the Terak Corporation 8510k, Which is based on the Digital Equipment LSI-11. For the current market, the Terak unit's price of $5500 to EDUCOM member institutions is highly competitive. Nevertheless, the rate of new announcements from the industry continues very high, and we believe that it is all but impossible to predict what hardware will provide the best cost/performance tradeoff for as long as even one year in advance. Of necessity, our search has concentrated on stand-alone microcomputers with graphic display capabilities, and with enough main memory and secondary storage to handle the extensive software and course materials with which we are working. We welcome inputs on this subject from other institutions, or from any vendor, and endeavor to keep EDUCOM informed of opportunities that seem advantageous. In addition to educational and communications applications, we are interested in word processing and business applications of the same machines.

[12] The following sections of this status report contain brief detailed summaries covering most of the topics just enumerated. If we haven't answered your questions yet, please try again with a phone call or letter. For those who already have our software system in use, we will soon be providing an automatic Tele-Mail facility on a dial-in basis. This should improve dramatically our ability to keep you informed and to respond when software difficulties arise.

[12] 2. The PASCAL based Software System

[12] Thus far, users who have received our first released system have copies of version 1.3, which was completed in mid August this year. We have ourselves been using version I.3c since early October. By the end of the December academic break, we hope to have a version I.4 available for distribution. The most

significant generally useful addition since the 1.3 release has been the screen-oriented editor. A major package for preparation of CAI programs, following the general philosophy of the University of California Irvine Physics Computer Development Project (PCDP), has been placed in operation on the Terak 8510k microcomputer. Except for some graphics materials within this package, it can be used on a wide variety of CRT screen display devices. Software more specifically oriented to the Terak machine is also available, and includes a character set editor (for the soft character generator), and a bookkeeping package for keeping track of student progress in a large Keller Plan (PSI) class.

[12] The software system is currently executed in a pseudo machine interpreter, which emulates a hypothetical real machine designed to handle PASCAL constructs efficiently. Our pseudo machine is similar in concept to the P-machine distributed by Wirth's group at Zurich, but we have made extensive changes to compress the PASCAL object code into a much smaller space than possible with the Zurich interpreter. The interpreter, and run-time support routines, currently occupy about BK bytes of main memory. The interpreter is in the native machine language of the host machine, and thus far has been coded by hand using the host's assembly language. All other code in the system is written in extended PASCAL.

[12] While the interpreted object code runs roughly five times slower than native code for the host machine, several factors allow our large system programs to run substantially faster than this would indicate. The strategy of code compression makes it possible to run relatively large programs without time consuming overlays. For example, the complete compiler occupies 20K bytes in a single overlay. Since the system is designed for frequent compile/go cycles associated with instruction, we have added several built-in procedures and functions to handle low level Operations needed frequently by the compiler. As a result, the compiler translates PASCAL source code at about 650 lines per minute on an LSI-11 with its clock set to 2.2 MHz. On a Lt MHz Z80, the compile speed will be slightly faster than this.

[12] Interpreter based versions of our system are flow running on 5 distinct processors, and two others are close to completion. Those operating include DEC POP-11's ranging from the LSI-11 to the 11/45, using either floppy disks or RKO5 disks for secondary storage. Versions for the 8080 and Z80 are operating in our laboratory, but more of that later. Sperry Univac Minicomputer Operations at Irvine is using the system on the V-75 and related machines. Another group at UCSD has the system running on the Nanodata QM-1. With support from General Automation, a version is close to completed on the GAZLO family of machines. National Semiconductor has an implementation nearly completed on the PACE microcomputer.

[12] The principal modules of the system as it will be distributed in the 1.14 release include the following:

PASCAL compiler File manager (capabilities similar to DEC's PIP) Screen oriented editor (cursor positioning, immediate updates) Line oriented editor (similar to DEC's RT-11 Editor) Debugger (single line execution, reference to variable contents) SETUP program (adapt system. to most ASCII terminals) BASIC compiler (ANSI standard plus strings) Operating System and user command interpreter PASCAL pseudo machine interpreter Linker program (for linking independently compiled program segments) Desk Calculator utility program

[12] Users of the Terak 8510A may, on request, also receive copies of the CAI package, and automated quizzes for the introductory textbook, as well as the bookkeeping package.

[12] Documents describing all of the above are available, and part of the release, but not all documents can be considered complete at this time. We distribute source and object code files on separate floppy disks formatted to be compatible with the IBM 37140 standard, with 512 byte blocks laid out in alternate 128 byte sectors according to DEC's standard. We have occasionally sent copies recorded directly on disk packs for the RKO5 drives. All other media are painful or impossible for us to handle, and no promises are made to use them. Those who order the full $200 release package will be sent both the documents, and printed listings of the source programs. Copies of the descriptive documents, amounting to approximately 150 pages, may be ordered at $10 each (checks payable to the "Regents of the University of California") to cover printing and handling costs.

[12] 3. Minimum Configuration

[12] In order to use the compiler, you need a total of at least 148K bytes of main memory, including the 8K bytes assigned to the interpreter. We use 56K bytes. Ideally, the interpreter should be completed re-entrant

and thus it should be possible to operate the interpreter from Read Only Memory. To date, the ideal has not quite been achieved, as none of our sponsors has yet insisted on that feature.

[12] At present, the system we use with students contains several built-in functions not needed for system development. The aggregate size of these functions is large enough to prevent compiling the compiler itself, or the operating system, even on a 56K byte system. Accordingly, we currently have two versions of the system, one for students, and one for system development. Within the next few months, we plan to add a means of configuring general purpose libraries for the system, and by that means expect to be able to return to a single version for all purposes. That single version should be practical to use in less than 18K bytes for some purposes.

[12] If you intend to compile on one microcomputer, and to executed object routines on others, the others can get by with as little as 1&K bytes of main memory if the operating system is not used. The resident portion of the operating system occupies about 8K bytes itself. This will undoubtedly be reduced as part of the libraries project.

[12] The system is designed to be used with standard IBM compatible floppy disks, Clearly it can be used with other varieties of floppy disks, or with other secondary storage media, with appropriate I/O drivers. The I/O drivers have proven to be one of our principal bottlenecks, and we make no promises in advance about supporting other devices. For DEC PDP-11 machines, the floppy disk drivers are assumed to be compatible with the RX-11, or with the Terak 8510A drives. Hard disks are assumed to be compatible with the RKO5.

[12] The system is normally supplied with the assumption that the user has a simple line-oriented ASCII terminal. The SETUP program can be used to configure control codes for more appropriate use of most CRT terminals. Copies of' the system supplied to users of the Terak 8510A make fairly extensive use of the special graphics and character generator facilities of that. machine.

[12] 4. 8080 and Z80 versions

[12] The Z80 version is now running on the Tektronix 8002 Microprocessor Development Aid system, for which Tektronix has supplied substantial support to the project. The 8080 version uses virtually the same source code as is used on the Z80, with conditional assembly altering certain passages in the source to substitute for a few of the extended Z80 instructions that proved useful.

[12] Release of the 8080/Z80 version of the system for other machines has been held up primarily because of the awkwardness of handling I/O. We currently have a Zilog Development System, a Processor Technology SOL system, and a Computer Power and Light COMPAL-8O system. The floppy disk provisions for each of these machines is non-standard. As a result, we have been forced to down-load programs via serial interfaces to get from the LSI-11 host machines used for development over to the new 8080 or Z80 based host, This has proven to be a very time consuming process, and a serious bottleneck in our work. Moreover, we are somewhat amazed to find that the Assembly of large programs on these machines runs almost a factor of ten slower than compilation of PASCAL programs that carry out similar tasks. Clearly, something has to give if we are to reach the objective of distributing PASCAL systems for more than a few 8080 and Z80 based machines.

[12] The solution to this problem that we now plan to use is based on the extensive market penetration of an operating system called CP/M, which is a product of Digital Research Inc. We have talked with many OEM and hobbyist users of the 8080 and Z80 who wanted to know when we would have the PASCAL system operating under CP/M. We then learned that CP/M is distributed in a package which assumes that most users will write their own I/O drivers. In effect, CP/M establishes a quasi standard for the interface between an 8080/Z80 operating system and its I/O drivers. With thousands of copies working in the field, CP/M seems to be far ahead of' the field in this area. Accordingly, we have decided to release the UCSD PASCAL System for 8080 and Z80 users in a form that will work with I/O drivers and bootstrap loaders developed for use with CP/M. This does not mean that our package will run under CP/F4. However, if CP/M runs on your machine it should be relatively easy to install the PASCAL system on that machine. We have been in contact with Digital Research on this concept, and they have offered to cooperate. If you do not have CP/M for your machine, the implementation package may be obtained from Digital Research Inc., Box 579, Pacific Grove, CA 93950 for $70. Since CP/M has been implemented on a very wide variety of

8080 and 280 based machines, there is a high probability that CF/N I/O drivers are already available from Digital Research or someone else for your machine.

[12] Alteration of our present interpreter to match the CP/M I/O calling conventions has proven to be very simple, at least on paper. We expect that some implementors of CP/M will have installed standard console input routines which automatically echo to the standard console printer or display device. This will necessitate a change, since our system uses both echoing and non-echoing console input. At this writing, the exact method to be used is under discussion. Barring some unforeseen calamity, copies of our system designed to run with CP/M I/O drivers should be ready for distribution by early January, 1978. The distribution medium will be IBM compatible floppy disks formatted in a manner yet to be finally specified. We will undertake to transform the system for other media and other formats, in general, only if a copy of the necessary hardware is available in our laboratory, and only if funds are available to pay for the extra conversion work.

[12] For many of the 8080 based machines we have seen, the most practical way to install our system will be to use 18K bytes of RAM augmented with BK bytes of ROM for the interpreter. Any additional RAM or RON required by the host processor system will also be needed.

[12] 5. PASCAL Extensions and Alterations

[12] We have attempted to implement faithfully as much as possible of PASCAL as defined in Jensen & Wirth's User Manual and Report. The principal extensions to PASCAL embodied in our system are related to STRING variables, Turtle Graphics, handling of disk files, Segment (overlay) Procedures, and several functions for support of the system itself. Alterations include a prohibition against passing procedure or function identifiers as parameters, restriction against GOTO out of a procedure, the addition of EXIT(<procedure-name>) to effect a normal exit from the procedure named in the parameter, and a change in READ applying to the interactive INPUT and KEYBOARD files. Further details than given in this section are given in our system release documents.

[12] Type STRING is a pre-declared record containing a character count followed by a packed array of characters. Built-in procedures and functions include LENGTH, POS(ition), INSERT, DELETE, COPY (i.e. extract), CONCATenate, SCAN, FILLCHAR, MOVERIGHT, and MOVELEFT. The last four of these also operate on conventional packed arrays of characters.

[12] Turtle Graphics describes a technique originated by Seymour Papert of MIT in which one can either MOVE a cursor (called the "turtle") an arbitrary number of screen units in the current pointing direction, or TURN an arbitrary number of degrees at the current position. A PENCOLOR procedure allows the line drawn by a MOVE to be either WHITE, BLACK, or NONE.

[12] The disk file extensions allow working with fixed length logical records corresponding to any legal <type>, which might typically be a RECORD data structure. GET and PUT operate normally through a window variable of the same <type>. OPENNEW creates a new file, OPENOLD opens a pre-existing file, and CLOSE allows saving or purging a file. SEEK (which will be distributed with the 1.11 system for the first time) allows random access to logical records within a file. SEGMENT Procedures are separately compiled and then linked into the host program using the LINKER. A Segment procedure is only loaded into main memory when it is entered for the first time, and its memory space is deallocated upon exit from the first invocation.

[12] READ(INPUT,X) is defined by Wirth as X: INPUT f; GET(INPUT); which we find to be extremely awkward for interactive use. Our solution is to place the implied GET before the implied assignment in the case of interactive files of type TEXT. READ operates as defined in Wirth's Report for other TEXT files.

[12] PACKED records on our system which fit within 16 bit fields are automatically packed and unpacked without explicit action by the programmer.

[12] 6. Introductory PASCAL Course and Textbook

Many of those inquiring about our system have heard about it through having seen the textbook "Microcomputer Problem Solving Using PASCAL" by the author of this note, published this fall by Springer Verlag. If you haven't seen a copy, they may be obtained from Springer at 175 Fifth Ave., New York City, NY 10010.

[12] The book is the basis for teaching the large attendance introductory computer science course at UCSD. This course comes close to matching the specifications for course CSI in the recently published curriculum recommendations from ACM's SIGCSE. The approach is non-numerical as far as practical, as a tactic to reach the many students who come to us with inadequate preparation in high school mathematics. The problem solving and programming approach taught is the same as we would teach even if all the problem sets were mathematics oriented. Because many problem examples and illustrations use our string and graphics extensions to PASCAL, the textbook currently assumes that the student will have access to a computer which runs under the UCSD PASCAL system. We will be glad to discuss the possibility of conversion to other software systems, but have very limited resources to apply to such conversions. There are several stand-alone microcomputers now being sold in large quantities on which our system would run, given a small conversion effort, and we would welcome support funds to pay for such conversions.

[12] Software in the form of automated quizzes is available with our system release for those who may wish to teach using the textbook. Each chapter in the book has a list of study goals for the students to achieve. Wherever appropriate, the quizzes test for mastery of the topics enumerated in the goals lists. The quiz programs have been implemented using a set of CAI primitive routines patterned after the well known DIALOG CAI system developed at U.C.Irvine by Alfred Bork and his colleagues.

[12] The introductory course is taught using Keller's Personalized System of Instruction (PSI). PSI has been found to be a more successful method of instruction than any other method commonly used in universities and colleges. This success is achieved, almost completely without conventional lectures, by using experienced students as Learning Assistants called "proctors". The characteristics of this method make us believe that it is possible to offer this course, or others constructed along the same lines, on a packaged basis for use at other institutions. A separate paper describing this possibility in detail called "Microcomputer Based Mass Education" is available from the writer of this status report.

[12] 7. Tele-Mail User Support Facility

[12] We have reached the point where it will be possible for us to begin operating a dial-in computer "mailbox" by early in the winter quarter. We have been using the Terak 8510A machines occasionally as intelligent terminals for exchanging messages via the large B6700 computer operated by the campus computer center. Our own Tele-Mail facility will use its own single telephone number reachable directly from the national dialed telephone network, or internally via the California state government telephone network. Paid subscribers to our software release will be notified when this mailbox facility is ready to be used.

[12] The mailbox will be operated primarily to serve users of our software system. It will provide notices of recent bug corrections, down-loading of program files (either source or object) where appropriate, notices on new additions to the software and new machines on which implementations have been completed, and other useful information from us to the users. It will also serve as a means for us to collect messages from specific users, and to answer them expeditiously, without the hassle of both parties having to be at their telephones at the same time.

[12] Through the use of block transfer software, the mailbox will make relatively efficient use of the dialed telephone network. We would like to begin immediately by offering a dial-in port at 1200 bits per second. However, the present state of confusion in the industry at that speed (which is the fastest one can use with acoustic couplers) leads us to move cautiously. We can and will install a port at 300 bits per second using the standard Hell 103A equivalent conventions. The system will answer an incoming call from an ordinary terminal by providing a brief summary of recent developments. It will otherwise expect a "handshake" from a special file transfer program that we will provide to users of our software package. This program will be the means of interchange based on efficient transfer of messages in the form of complete files. If you wish to send an ordinary text message to us, you will prepare the message using either of the editors built into the system. Only after the message is complete will you need to make the telephone connection.

[12] 8. Forthcoming Improvements

[12] As mentioned earlier, our next significant improvement in the software will be a more flexible system allowing libraries of programs. One of the main reasons for doing this will be to allow the software to be configured to make efficient use of main memory in cases where the user does not need all of the built-in

facilities. For example, we have no need for turtle graphics when compiling large system programs.

[12] One of the long awaited features of the new library system will be an arrangement allowing mixture of PASCAL procedures with Assembly language routines and/or procedures compiled directly to the native code of the host machine. The necessary assemblers and code generation will come somewhat after the library system is operational. If all goes well, the library system should be ready to distribute during the winter quarter of 1978. The assemblers and native code versions of the compiler will come somewhat later as time for the necessary work permits.

[12] Many people have asked whether we have in mind extensions to support Concurrent PASCAL, or similar facilities to allow independent processes running concurrently. This is something we would like to do eventually, but our current resources do not allow making definite plans in this area.

Pre-I.2

[1] What was to become UCSD Pascal began in October 1974, by Kenneth Bowles and Mark Overgaard.

[2] While Roger Sumner was a junior at UCSD's Revelle College, 1974, Dr Bowles asked him to participate in the Project's system software design and development. Porting of Urs Amman Pascal compiler to the UCSD Pascal environment, and the design and implementation the PDP-11/LSI-11 P-Code interpreter.

[1] Niklaus Wirth's *Pascal User Manual and report* was quickly adopted at more than 300 Computer Science departments . Urs-Ammann's P-machine allowed (almost) instant Pascal on diverse machines. Pascal was a big influence on adding Science to Computer Science.

[1] The p-Machine has similarity to B6500 stack machine hardware. Used assembly language to implement P-machine on the PDP-11. Compiled Pascal on B6500 to write primitive UCSD Operating System, and to port Ammann's compiler. Ammann's Pascal Compiler re-compiled for UCSD p-Machine. Students developed the Editor, the Filer, etc, on the PDP-11 using Pascal

Which students? When? Name names. Date dates.

[1] Bowles persuaded Terak to build cheaper LSI-11 small computer. The Terak UCSD design widely adopted at other universities. (*when?*)

[1] The first port of the UCSD p-System was to the Z80, and it was a revelation. Peter Lawrence and Joel McCormack worked on the first trial demo in early 1976. First step... showed Sumner's Op System seemingly worked. Plugged in floppy disk with Kaufmann's Editor, All Pascal software worked on Z80 as on PDP-11, with no change. Portable software.

[1] Software was sold with a $15 license fee, to support project infrastructure.

[1] There was a need to extend the Pascal language definition to accommodate some features of small machines. This was not met with universal approval.

I.2 (*date?***)**

[7] released in-house at UCSD. Not yet self-hosting?

I.3 (Aug-1977)

[7] released at UCSD, also released to a few other UC campuses. All parts of the system still contained in SYSTEM.PASCAL.

The first "working" version of the interpreter, for reasonably loose definitions of the term "working", was I.3a. This was the p-machine that was used on the PDP-11/10s in the lab. [Keith Shillington, Pers. Comm., 2010-May-31]

I.4 (Jan-1978)

[7] general release. System divided into separate filer, editor, compiler.

I.5 (Sep-1978)

[7] UNITs developed, compiler now able to compile them. Editor now able to copy from other files. Filer how has wild-carding. Many other features.

I.5 has sub-versions a through f (several debugging efforts).

II.0 (Jan-1979)

[7] Second p-machine version. Division of single Remote I/O channel into separate input and output channels. Not much difference in organization from I.5

Swansong (1979-Jul-09)

[14] Events have once again overtaken us resulting from continued rapid growth in interest throughout the industry in UCSD Pascal***. Once again we have to apologize for the long delay since our last newsletter. This time, the growth has forced major changes in the nature of the Project. As a result, this newsletter will probably be the last one distributed from UCSD to our full mailing list.

[14] Commercial Licensing of UCSD Pascal™

[14] Readers familiar with the recent progress of our Project will recall that our top priority objective is to promote the concept of machine independent software. To move a large and complex applications program from one machine to another, we have found it vastly more practical to move the entire software system, i.e. UCSD Pascal ™ than to re-compile the applications program using just the compiler for the same high level language on each machine. The reason for this is that practical use of a programming language, no matter how well standardized, demands uses of operating system facilities. These facilities often must be reached using language constructions that fall outside the language standard specifications. Our experiences in this area have been so successful that we have felt obligated to pursue a course whereby the same UCSD Pascal System can be made widely available on machines of many different designs.

[14] NOTE: “UCSD Pascal” is a trademark of the Regents of the University of California. Use thereof in conjunction with any goods or services is authorized by specific license only, and any unauthorized use thereof is contrary to the laws of the State of California.

[14] As a secondary objective, we are, of course, helping to promote the wider use of Pascal. Again, the objective of program portability demands language standardization. During the last year tremendous progress has been made toward international standardization of Pascal. The work is being led by a committee of the British Standards Institute, who have issued a draft for a new standard definition of Pascal. This draft describes virtually the same Pascal as described in the original Report issued by Niklaus Wirth. The new draft clears up a large number of ambiguities and inconsistencies, in Wirth’s definition, making the language definition easier to understand. In the United States, a joint committee of ANSI and IEEE is actively participating in this effort. In view of the widespread use of UCSD Pascal™ we have felt obliged to support a version of Pascal which agrees with the standard language as closely as practical. In common with most other implementations of Pascal, we have made a few carefully chosen extensions to the standard language. Our “base” language does differ slightly from the new draft standard, and efforts are under way to correct these differences. We are disappointed that it seems very unlikely, in view of the real world politics of standardization, that a widely used standard will soon emerge on extensions to Pascal which are needed for some common applications.

[14] Beginning about a year ago, these objectives led us to arrange for commercial licensing of the UCSD Pascal System under circumstances that would discourage advertising as “UCSD Pascal” any version of the software not functionally identical to versions issued by the Project. This required a close working relationship with many manufacturers, to assure correct installation of the System on their equipment. A substantial aggregate level of income to the Project was required to pay the student employees engaged in the installation of the System on various different equipment models. By the end of the Fall quarter we were beginning to learn how to conduct this activity reasonably efficiently.

[14] Because of legal constraints on University of California activities, the University administration directed that the Project either cease operations or find an outside licensee to handle the routine maintenance, user support, sub-licensing, and installation of the System on additional machines (as well as on all those already licensed). It soon became apparent that other legal limitations would make it impractical to use a not-for-profit outside licensee. For other reasons, well established practices of the University of California (as well as many other universities) indicated the use of just a single outside profit making firm as the University’s “Sole Licensee” for support services covering UCSD Pascal™. Had several firms been licensed to handle sub-licensing, under a competitive arrangement, other laws would have made it impossible for the University to compel the several firms to release and maintain the same version of the UCSD Pascal System. With the sole licensee, the program portability objective is optimized

because the the licensee will distribute a common version of the system for all implementations.

[14] In the choice of a sole licensee, many other considerations were taken into account. We felt it necessary to choose a well established software house whose business history indicated a respect and understanding for the systematic programming principles on which Pascal is founded. The firm had to be large enough, and financially strong enough, that the University could reasonably assure the user community that commercial quality support services would be available indefinitely to back commercial distributors of UCSD Pascal™. Without this assurance, many of the commercial distributors were already showing signs of creating their own software support staffs − with the inevitable result that the many distributed versions would soon differ from the University's version. (While we know there are many shortcomings in the design and implementation of UCSD Pascal™, inter-machine portability of large application programs demands that all versions be the same in spite of those shortcomings.) The firm chosen also had to be small enough to minimize the layers of bureaucracy in University communications needed to oversee the work of the sole licensee. To foster continued close working relationships with the surviving research and education components of the Project within the University, preference was given to firms willing to locate the principal support office for UCSD Pascal™ in the close vicinity of the UCSD campus.

[14] Within these constraints, one of our major concerns was to find a firm willing to pursue the support and sub-licensing of UCSD in a way that would lead to widespread availability of portions of the System to students in colleges and schools at the lowest possible prices. It must be emphasized that this objective cannot be achieved by releasing all details of the current UCSD Pascal System into the public domain. If there were no copyright protection and commercial publication of college textbooks, and texts were required instead to be released to the public domain, there would be no system of mass education at the college level. Similarly, widespread availability of UCSD Pascal to college students demands a distribution approach that in some cases will resemble textbook publishing. Moreover, the UCSD Pascal System will not survive as an important tool for education unless it is also used extensively for commercial computer applications in many different ways. A substantial staff is needed for support and maintenance work necessary to assure that the quality of the UCSD Pascal System will improve, or at least not decline, as it evolves. The sole licensee firm must therefore be able to pay that staff and make a reasonable profit.

[14] As a result, the best we could do (for students and individual users) was to seek a sole licensee willing to work with companies interested in "publishing" smallish configurations of the UCSD Pascal™ System in large quantities at modest prices. Because of legal constraints, the University is unable to specify the pricing policies of the Sole Licensee. We sought a sole licensee willing to price its services within the range of the many small companies who wish to distribute UCSD Pascal™. Beyond this, the sole licensee's pricing will depend upon feedback from the marketplace. We feel that the interests of the small companies coincide with the University's objective of promoting program portability via machine independent software. Experience with the larger companies, who have so far indicated an interest in distributing UCSD Pascal™ has been that they usually seek licensing arrangements giving them complete freedom to modify as they wish. Therefore, it appears that the interests of the small distributor firms, and of individual users in general, will be favored if the sole licensee can base a strong and growing business on licensing arrangements which reinforce the machine independence concept. The sole licensee can only do this by earning a reasonable margin of profits from work with all sub-licensees, both small and large.

[14] Given the large community now interested in UCSD Pascal™, a brief comment on your opinions seems indicated. We understand that not all users of UCSD Pascal™, whether individuals or organizations, are enthusiastic about the changes described in this section. We hope that readers will understand that we too are frustrated about some aspects of the new licensing arrangements. The greatest frustration, by far, results from having to satisfy the dozens of laws and regulations that apply at the federal and State of California levels. The new licensing and support arrangements are the result of more than six months of negotiation, and examination of almost every aspect of the Project and its distribution of UCSD Pascal™. The new arrangements have been negotiated with extensive participation of the University's central administration and legal departments. The status of public access to the software products of the Project, and the stated objectives for the new licensing arrangements, are the best. we have been able to achieve in making UCSD Pascal™ a community wide resource, within all the imposed constraints. We appeal to readers to accept the new arrangements, and to assist us in taking advantage of the machine independent

aspects of the UCSD Pascal™ System, and its role in helping to make the Pascal language more widely used.

[14] *SofTech Microsystems Inc.*

[14] The sole licensee chosen by the University is SofTech Microsystems Inc. (abbreviated as “SMI” in parts of this newsletter). SMI is a newly formed subsidiary of SofTech Inc, of Waltham, Massachusetts. The principal business of SMI will be to provide support, maintenance, sub-licensing, and other services related to the UCSD Pascal™ System. They may be reached at:

Until 1 September, 1979	After 1 September, 1979:
SofTech Microsystems, Inc. P.O. Box 28010 San Diego, California 92128	SofTech Microsystems, Inc. 9994 Black Mountain Rd., Bldg 3 San Diego, California 92126
Tel: (714)741-1353 (temporary)	Telephone pending

[14] Not the least of the reasons for selection of SofTech as the sole licensee was their extensive expertise in managing complex software projects. As the Project began to support versions of the UCSD Pascal™ System adapted for many different processors and machines, it became apparent that the control of all those versions to operate identically was getting out of hand. SofTech has a major “Software Engineering Facility” for microcomputers, called the “MSEF”, which will be used starting immediately to help bring some order out of the chaos that has developed from the (necessarily) casual management environment of a university project mainly staffed by students. In addition, SofTech’s management agrees with our commitment to make the Pascal base language supported by the UCSD Pascal™ compiler conform as closely as possible to the new (draft) international standard for Pascal. The MSEF and SofTech experience will help considerably to complete this task expeditiously.

[14] As this newsletter was being prepared, most of the initial staff of SMI was just beginning to work for the firm. Roughly half of the initial staff consists of people who, until recently, have worked for the Project as student employees of the University. Among the principals of the Project, Mark Overgaard has joined SMI as a full time employee. This writer, as director of the Project, remains as a full time employee of the University with no financial interest in SMI, and with no employment status with them. All parties concerned would have preferred that both Mark and this writer could have taken split appointments, partly at SMI and partly at the University, in order to maintain the closest possible working ties between the Project and SMI. This has proven unworkable because of the California Conflict of Interest Code for employees of public institutions. Nevertheless, every possible effort is being made to ensure that the close working relationship between the UCSD Project and SMI will continue.

[14] *Transition to Support of UCSD Pascal™ by SofTech Microsystems*

[14] After a brief transition period, SMI will henceforth be responsible for all (sub)licensing for distribution of the UCSD Pascal™ System, or any of its components, to end users. The Project (i.e. UCSD) will distribute copies of portions of the System under the present catalog arrangement until the close of business on 14 August, 1979. Orders not requiring special handling will be processed by UCSD in the order they arrive. Any orders not yet processed by the available staff on the 114th of August will be turned over to SofTech Microsystems for whatever follow-up they consider most appropriate. Users concerned with this switch should understand that the nucleus of Project employees who have run the support group will become employees of SMI on 15 August. Thereafter, the Project will not be able to accept incoming orders, and there will be no support staff who might process such orders. Some orders for the I.5 version of the System were accepted by the University before the current catalog based distribution mechanism went into effect in January this year. Obviously, the hold-over offer of $100 credit against future catalog orders (presented to those who ordered I.5 before the catalog system went into effect) will have to expire on 14 August. We apologize about the short notice, but circumstances have made it impossible to do any better.

[14] All existing individual licenses for use of the UCSD Pascal™ System contain a clause stating that the license holder will have unlimited rights to the licensed materials after two years, unless the University terminates the license before that time. As a result of the negotiations described in the previous section, the University is now obligated to terminate all of the existing individual licenses. We hasten to assure

licensees that they will be offered a replacement for the licenses, with rights similar to those provided in the original licenses, but lacking the reversion to unlimited use after two years.

[14] All sub-licensing of distributors of portions of the UCSD Pascal™ System was transferred to SMI as of early June, 1979. It is expected that amicable arrangements will soon be made with virtually all firms who currently are licensed by the University for distribution of UCSD Pascal™, such that replacement licenses will be negotiated with SMI.

[14] In preparation for the new licensing arrangement, the term “UCSD Pascal” has been made a legal trademark of the University of California. The sole license agreement requires SMI to certify that sub-licensees, who use the term “UCSD Pascal” in advertising or describing the software they distribute, do so only when the distributed software meets specified functional tests. The tests are designed to assure that all software identified as originating from the UCSD Pascal Project will function in the same way, thus enhancing the portability of application programs. For purposes of the transition to sub-licensing solely by SMI, distributors currently licensed by the University will be regarded as having passed the certification tests, but only for the version(s) of the System already licensed.

[14] At this writing, SMI is busy preparing an announcement of services to be offered, and pricing for those services as well as sub-licenses. Since many of SMI’s employees have been hired at the going high salary rates for system programmers, readers should not be surprised to find that SMI’s prices will generally be higher than those previously offered by UCSD. In effect, these higher prices are an unavoidable necessity if good quality support services are to be available for the System on a continuing basis.

[14] *The Great Version Number Fiasco*

[14] If you order UCSD Pascal™ from a commercial distributor you may find that it is Version I.4, I.5, II.0, II.1, or III.0. Some *early* versions are *based on* but not identical to Versions I.3 or I.4. We have heard of vendors selling Version III.0 as better than II.1, and vice versa, whereas there is some truth in both claims! Herewith a brief recounting of how this mess arose, and how it will probably be cleaned up.

[14] Version I.5 was first released from UCSD in the fall of 1978. It differed from I.4 mainly by adding the facility to compile and use independent collections of routines called “Units”. This facility makes it possible to provide a large library of service programs written in Pascal (or Pascal mixed with Assembly Language), and to use them without re-compilation.

[14] During the summer and fall of 1978, the P-machine interpreter for UCSD Pascal™ was implemented to run on several additional processors, notably the 6502, 6800, 9900, and GA-16. It was hoped that the interpreter for these processors could incorporate design changes to make the P-machine generally compatible with a wider class of processor architectures. During the same period, the Project agreed to work with Western Digital on the development of the microprocessor W/D is now selling as the “Pascal MicroEngine”. This microprocessor is programmed entirely in Pascal, with no machine level or assembly language being supported. This made it necessary to augment UCSD Pascal™ with provisions for concurrent processes, so that interrupt routines could be written. At the same time that the P-machine interpreter was being revised for this purpose, changes were also made to make the P-machine more compatible with the extended address ranges of the new generation of 16-bit microprocessors. Because the implementation work for the 6502, 6800, 9900, and GA-16 overlapped with the revision work on the P-machine interpreter, *some* of the revised P-machine features found their way into the interpreters for those processors. Initially, it had been hoped that the first distributed versions for these processors could take advantage of the full set of P-machine improvements. By the end of 1978, it became clear that this was not feasible, since the system software to support all of ’ the new P-machine features would not be completed for several more months. To reduce the chaos, we decided to establish as the single (interim) P-machine, for all processors other than the MicroEngine, the version already implemented on the 6502, 6800, 9900, and GA-16.

[14] The interim interpreter was designated as II.0. It was first released during February, 1979. Software for II.0 was virtually the same as Version I.5 except for corrections of reported bugs in I.5.

[14] Version II.1 has evolved from II.0 as a way to make the System easier to use on machines with mini-floppy disk drives. Though I.5 and II.0 provide much greater flexibility than Version I.4, because of the new Units facility, the linker required to work with Units imposes a heavy overhead burden (processing

time, space in memory, space on the disk). Version II.1 adds “Intrinsic” Units to the facilities of II.0. An Intrinsic Unit may be used without resort to the linker. This eliminates the need to retain a copy of the linker on one’s principal working disk, and eliminates the substantial time delay associated with the linking process after every compile. (It is still necessary to use the linker to incorporate assembly language routines into a Pascal program Unit in the form of EXTERNAL procedures.) At this writing, II.1 has been released for the Apple II computer, and is being distributed by Apple Computer Inc. (not by the Project!). Work is well under way to convert all other implementations supported by the Project, apart from the W/D MicroEngine, to II.1 within the next several months. This conversion has been delayed somewhat by the shift of licensing and support to SofTech Microsystems.

[14] Version III.0 is the initial version developed with enhancements for concurrent processing. It is the first version to be available on the Western Digital MicroEngine, which is now being delivered to customers. The Pascal language facilities of III.0 are largely those of the I.4 version of the System, and thus III.0 still lacks the Units facility of I.5, II.0 and II.1. Work is under way to augment the III.0 version to include all of the facilities of I.5, II.0 and II.1.

[14] SofTech Microsystems will now be responsible for releasing a single new version of the System which will provide the beneficial features of all previous versions. This new version, tentatively called Version IV, will probably not be available until 1980. It is not yet clear whether an interim Version III.1 will be issued for the W/D MicroEngine, to bring the System on that machine closer to the others before Version IV is completed.

[14] We emphasize that nothing in this summary constitutes a promise of delivery of any new version of the System. SofTech Microsystems intends to issue a description of their plans by 1 September, 1979.

[14] *User Group*

[14] There have been frequent suggestions or requests that a UCSD Pascal™ user’s group be established. We have been very sympathetic to this concept, but have felt it inappropriate for us to organize such a group ourselves. Moreover, it has not been possible to allocate University resources to support such a user’s group.

[14] Now SofTech Microsystems has indicated a willingness to encourage the creation of a user’s group, and to supply some resources and assistance to that end. Readers interested in assisting with the formation of a user’s group should write to SMI (not UCSD!).

[14] *Update on Processors and Implementations*

[14] As of this writing, implementations of current machine independent versions of the UCSD Pascal System have been licensed for distribution on the following processors:

6502
6800, 6809
* 8080, 8085, Z80
9900
* General Automation GA-15
* PDP-11, LSI-11
Western Digital MicroEngine

[14] Note that the Project distributes copies to individuals only for the processors noted with “*”. SofTech Microsystems will announce its own plans on individual distribution for any of the processors in this list. (Remember that their distribution starts on 15 August, 1979).

[14] Licensed versions, though not fully compatible with those supported by the Project, are available for the following:

Alpha-Micro 100
Data General NOVA
Nanodata QM-1

[14] An experimental version has been implemented on the Hewlett Packard 9835 desktop computer, and another on the Lockheed Sue. Several bit-slice microprogrammed implementations have also been reported.

[14] The Project has a version nearing completion for the 8086 processor. This activity is currently delayed for the lack of hardware on which to complete testing.

[14] *Intensive Training for UCSD pascal™*

[14] Many people who communicate with us have asked when/whether a course would be made available on Pascal, and on the UCSD Pascal™ System, for people who already can develop programs in other languages. In cooperation with Integrated Computer Systems, Inc. (the “Learning Tree” people), an intensive 14-day short course is now being developed for first offering in October this year. About half of the available time will be used for guided “hands-on” use of the UCSD Pascal™ System on small microcomputers − probably Apple II’s. The ICS mass mailed announcement on this course should be arriving within the next few weeks. If you don’t already receive regular mailings from ICS, write to:

Integrated Computer Systems, Inc.
Box 5339,
Santa Monica,
California 90405

[14] *Future Project Plans*

[14] While all of the Project’s regular services in support of the UCSD Pascal™ System are being transferred to SofTech Microsystems, we expect to continue a substantial level of research and development, and education activities at UCSD. During the next year or two, we anticipate that only a small part of the financial support for these activities will come from royalties paid to the University under the sole license agreement with SofTech Microsystems. Virtually no support will come from regular University of California operating budgets. If the activity is to continue, the remaining support will have to come from other sources. The principal sources available to us are federal research grants or contracts, and grants from industrial firms under an “associates program”. The potential sources for federal grants or contracts are very limited at this time, and only one small contract is expected during the next academic year.

[14] Industrial associates programs are often the primary source of support for engineering oriented research projects at many universities. Under such a program, each of many member companies contribute each year an amount of money comparable to the cost of supporting the work of one graduate student. Several firms have already expressed an interest in joining such a program for the UCSD Pascal™ Project. We have been effectively unable to move ahead on organizing the UCSD Pascal™ Associates program until the licensing details described in this newsletter had been worked out. A prospectus will be sent shortly to firms known to have an interest in the Project. Inquiries would be most welcome from others. We are particularly grateful to Philips Research Laboratories, who have contributed without waiting for the organizing papers.

[14] Among the distinct activities, currently ongoing in the Project at UCSD, and for which financial support is being sought, are the following:

[14] a) Native code generation from Pascal. An experimental system is nearing completion for translating the P-code output of the UCSD Pascal™ compiler into the native machine language of the host processor. The approach being taken will lead, hopefully, to the implementation of code generators for different host processors with minimal effort. The earliest test cases will be the PDP-11, the 8080 and the 6502. The plan is to allow a programmer to designate which routines (procedures and functions) in a Pascal program will be translated to execute directly in native code, rather than in the P-code of the P-machine interpreter. Programs with mixed P-code and native code are already being used extensively, with the native code being generated at present only by the Assembler associated with the host processor. Whereas the assembler approach makes a program no longer portable from machine to machine, the generation of native code from Pascal should maintain the portability while giving the execution efficiency of native code.

[14] b) Distributed processing. During the next few months, collections of microcomputers running UCSD Pascal™ will be interconnected in several different “party line” bus configurations. One method, that has been studied extensively, uses high speed serial transmission. It allocates transmission rights to the stations connected to the network according to full character slots timed within a cycle of about one second duration. Another method uses commercially available components for interconnection over an 8-bit

parallel data bus similar to the IEEE 488 General Purpose Instrumentation Bus. One objective is to find an interconnection method (both software and hardware) whereby mixed microcomputers, dumb terminals, and large machines can communicate with each other within a large building complex at minimal cost. Another is to learn how best to distribute the processing resources of a collection of microcomputers, some of which will serve special functions (e.g. database access, number crunching, printer control, ...), while others will be used primarily for access to the network. In the course of this work, several concepts of a distributed operating system will be tested.

[14] c) UCSD Pascal™ Operating System improvements. The limitations of the present operating system are well known, and many ideas have been advanced on how it could be improved. Work currently going on should result in a far more general approach to handling the disk directory, in spite of the physical limitations of the typical microcomputer on which the System runs. Beyond this, there are several possibilities for designing multi-user versions of the UCSD Pascal™ System.

[14] d) Software Tools. The pre-compiled Units facility of current versions of the UCSD Pascal™ System permits the implementation of large libraries of frequently needed routines. These routines can be used by a programmer as if they were extensions to the list of standard procedures and functions defined as part of the Pascal language. Only those library files actually needed by the programmer must be present at compile time and execution time. Work now starting is seeking an organized basis for designing the many separate Units that a typical applications programmer may want. Units are being implemented for applications ranging from the authoring of CAI packages to database management and interactive data capture.

[14] e) Education Packages. The authoring package for creating Computer Aided Instruction materials is being thoroughly revised making used of the Units facility. We expect this package to continue evolving as experience grows in use of the existing introductory materials on Pascal programming, and as other CAT authors reports on their experiences. With the expected installation this fall of a distributed processing network connecting the microcomputers we use for regular teaching, experiments will begin in assisting students via remote communication from a teaching aide at a separate microcomputer.

[14] In general, we expect that software resulting from any of this work, that turns out to be of good enough quality to distribute to others, will be made available via the licensing arrangement with SofTech Microsystems. As with the software already turned over to them, we expect SMI to add substantial value to the new products of the Project by turning those products into commercially maintainable form, and by providing continuing maintenance, support, and distribution services. All inquiries regarding distribution schedules should be directed to them (at the address given earlier in this newsletter), as the Project will retain very minimal staff resources to answer correspondence in the volume it has been received in recent months. Naturally, the Project does intend to provide pre-release programs, documentation, and other materials on an informal basis to sponsoring organizations, particularly those who are members of the UCSD Pascal™ Associates program mentioned earlier.

SofTech Microsystems

[13] *Electronics*, 1979-Aug-16, p. 33: A notice that SofTech has acquired control of UCSD Pascal.

II.1 (*date?*)

[7] INTRINSIC units appear: linked directly from library at run-time, rather than being copied into the code file. System now has 32 segments rather than 16 (this change was not carried over to any later versions). UNITS can be declared as resident or non-resident.

II.1x7b More swapping capabilities added: screen handlers and others can be made to stay on disc, loaded only when needed.

SofTech Transition (*when?*)

[1] UC Tax Status Sealed Our Fate in early 1979, we were too successful. Though making no profit, income exceeded $1m per year. UC files no income tax return... but only if all income is from Teaching, Research, or Public-Service. UC feared UCSD Pascal “unrelated-business” income would trigger IRS demand for tax return for all of UC.

[1] License was to be given to an outside for-profit vendor, and stop licensing from UCSD itself. SofTech MicroSystems was the licensee, initially staffed by graduates of UCSD Pascal project.

III

[1] The UCSD Pascal Micro-Engine was by Western Digital Corp, who built hardware for UCSD Pascal. Team led by Mark Overgaard did the P-machine microcode. Cited initially as proof of special CPU design advantages. Barry Smith at Oregon Software soon proved that clever compiler design made Pascal code faster for the same chip with LSI-11 microcode.

[7] Volition Systems is a company formed by Randy Bush, of the original Pascal project at UCSD. From it came the first truly reliable versions of III (for the MicroEngine, of course). It also produced the first version of the Advanced System Editor.

[7] Among the changes introduced in version III is the first appearance of parallel processes, and the routines and types needed to control them.

[7] III.E1 Third p-machine version, by Western Digital Inc. Developed for the Western Digital MicroEngine, (thus far) the only hardware p-machine. Some architectural and philosophical changes from previous versions. Bugs in implementation of some p-codes.

[7] III.F Some debugging, minor improvements in I/O. Single character typeahead only. Sub-versions are F0 and F1.

[7] III.G0 More debugging. NO typeahead whatever.

[7] III.H0 More debugging. Typeahead restored, but not reliable. Versions from Western Digital, from Volition Systems, and for PDQ.

[7] III.H1 By Volition Systems for the PDQ-3. Much more reliable than other versions of III.

[7] III.H1 IME,

[7] III.H3 More debugging. System reliability improved, at cost of increased system size and reduced available memory.

IV

[7] Most recent p-machine change. Extensive changes in both p-machine instruction set and Pascal system architecture. SofTech MicroSystems licenses various companies (such as IBM and Xerox) to bring up IV.0 implementations on their machines.

[7] IV.1 Under license from SofTech MicroSystems, several companies (such as Sage Technology in Nevada, and Network Consulting Incorporated in British Columbia) bring up IV.1 for various machines:

- NCI releases a version for the IBM PC more powerful, faster, and more reliable than that released by IBM.
- Ticom Inc. releases a version for the DEC Rainbow 100.

Advanced Systems Editor now available under IV. Scheduling of parallel processes debugged.

Legacy

Authors

Gillian M. Ackland

[3] *what? when?*

[4] Acted as Editor for the compilation of The Proceedings of the July 1979 UCSD Workshop on Pascal Extensions.

[5] Listed as an editor, UCSD Pascal II.0 User Manual.

Mark Allen

[3] *what? when?*

[8] Lists him as working with Richard Gleaves on the 6502 interpreter. This contradicts Richard Gleave's statement that it was Mark Overgaard. Were all 3 working on it?

S. Dale Ander

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

Lucia Bennett (now Yandell)
: [5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

  [5] Listed as a software author, UCSD Pascal II.0 User Manual.

Marc Bernard
: [5] Listed as a software author, UCSD Pascal II.0 User Manual.

Kenneth Bowles
: Director. It isn't clear whether or not he wrote any UCSD p-System code.

  [1] What was to become UCSD Pascal began in October 1974, by Kenneth Bowles and Mark Overgaard.

Randy Bush
: [7] Volition Systems is a company formed by Randy Bush, of the original Pascal project at UCSD.

Raymons S. Causey
: [5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

Charles Chapin
: [5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

J. Greg Davidson
: Mentioned in the BASIC sources, 11-Apr-1979

  Mentioned in the Setup sources, II.0 [D1] 11-Apr-1979

  Mentioned in the Yaloe sources, Jun-1977.

  [5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

  [5] Listed as a software author, UCSD Pascal II.0 User Manual.

Barry Demchak
: [3] mentioned

  [5] Listed as a software author, UCSD Pascal II.0 User Manual.

Gary J. Dismukes
: Mentioned in the LIBMAP sources, Mar-1979

  Mentioned in the Librarian sources, Mar-1979

  [5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

  [5] Listed as a software author, UCSD Pascal II.0 User Manual.

Julie E. Erwin
: [3] mentioned

  [5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

  [5] Listed as a software author, UCSD Pascal II.0 User Manual.

Shawn M. Fanning
: Mentioned in the Compiler sources, 1976..1979

  [5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

William P. Franks
: Mentioned in the Assembler sources, (?) Sep-1978

  Mentioned in the Disassembler sources, II.0 Sep-1978

  [5] Listed as a software author, UCSD Pascal II.0 User Manual.

Karen Fraser
: [3] *what? when?*

Richard Gleaves

[3] spent the summer of 1978 working (with Mark Overgaard) on-campus at UCSD writing the 6502 interpreter that later became the basis for Apple Pascal. (We were paid UCSD's standard student "junior coder" wage of $5.50 per hour.) Trivia detail: the interpreter was developed (and thus the first 6502-based Pascal system booted) on a Rockwell box.

[3] At the end of the summer 1978 Bill Atkinson started showing up in the lab, and he worked closely with Mark Overgaard to get the thing going on the Apple II. Later Bill Atkinson offered Mark Overgaard and Richard Gleaves jobs up in Cupertino, but they both turned him down because they both wanted to stay in San Diego.

[3] Later worked with Barry Demchak at ACD to co-develop the AOS (Advanced Operating System) variant of UCSD Pascal.

C. Richard Grunsky

[3] *what? when?*

Albert A. Hoffman

Mentioned in the Compiler sources, 1976..1979

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

Robert J. Hofkin

Mentioned in the LIBMAP sources, Sep-1978

Mentioned in the Long Integer sources, Jun-1977

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

Richard S. Kaufmann

Worked on the Editor, 11-Oct-1978, 10-Dec-1978

Mentioned in the Yaloe sources, Jun-1977, 7-Oct-1977, 9-Feb-1978.

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

Mary K. Landauer

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

Nancy Lanning

[3] *what? when?*

Peter A. Lawrence

[3] mentioned

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

J. Raoul Ludwig

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

Joel J. McCormack

[3] mentioned

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

Robert A. Nance

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

Mark D. Overgaard

[1] What was to become UCSD Pascal began in October 1974, by Kenneth Bowles and Mark Overgaard.

[6] Unlike most of the contributors, who were undergraduate students, Mark Overgaard was a graduate student.

[3] spent the summer of 1978 working (with Richard Gleaves) on-campus at UCSD writing the 6502

interpreter that later became the basis for Apple Pascal.

[3] At the end of the summer 1978 Bill Atkinson (from Apple) started showing up in the lab, and he worked closely with Mark Overgaard to get the thing going on the Apple II. Later Bill Atkinson offered Mark Overgaard and Richard Gleaves jobs up in Cupertino, but they both turned him down because they both wanted to stay in San Diego.

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

David A. Reisrier
[5] Listed as a software author, UCSD Pascal II.0 User Manual.

Bruce Sherman
[3] *what? when?*

Keith Allan Shillington
Mentioned in the Yaloe sources, 11-Aug-1977, 13-Sep-1977.

[2] Collected, Edited and distributed the copies of documentation and software for UCSD Pascal in it's early days on campus. Moved with the system to SofTech MicroSystems, and stayed with it for just over a year.

[3] spent the summer of 1978 working (with Richard Gleaves) on-campus at UCSD writing the 6502 interpreter that later became the basis for Apple Pascal.

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

[5] Listed as an editor, UCSD Pascal II.0 User Manual.

David A. Smith
[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

David M. Steinore
[5] Listed as a software author, UCSD Pascal II.0 User Manual.

George Symons
[3] *what? when?*

Roger T. Sumner
Mentioned in the System sources, Jan(?)-1977.

Mentioned in the Compiler sources, 1976..1979

Mentioned in the Filer sources, I.3 Jan(?)-1977.

Mentioned in the Librarian sources, I.5 Sep-1978

Mentioned in the Linker sources, I.5f Jan-1978, II.0 1-Mar-1979

Mentioned in the Long Integer sources, Aug-1978.

Mentioned in the Yaloe sources, 13-Sep-1977, 24-Sep-1977.

[2] Roger had been involved with the UCSD Pascal Project from it's 1974 inception. While Roger was a junior at UCSD's Revelle College, Dr. Bowles asked him to participate in the Project's system software design and development. Specifically, Roger's contributions to UCSD Pascal include the porting of Urs Amman Pascal compiler to the UCSD Pascal environment, and the design and implementation the PDP-11/LSI-11 P-Code interpreter and device drivers, the UCSD Pascal Operating System, Pascal language intrinsic functions, Filer utility and Linker utility.

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

Steven S. Thom(p)son
Mentioned in the Filer sources, I.5 Jun(?)-1978, II.0 Jan(?)-1979

[5] Listed as a software author, UCSD Pascal II.0 User Manual.

John VanZandt
Compiler (?)

Dennis J. Volper
Mentioned in the Assembler sources, 27-Sep-1978

[5] Listed as a documentation author, UCSD Pascal II.0 User Manual.

Bavid B. Wollner
[5] Listed as a software author, UCSD Pascal II.0 User Manual.

References

[1] http://www.jacobsschool.ucsd.edu/Pascal/ppt/KenBowles.ppt

[2] http://www.jacobsschool.ucsd.edu/Pascal/bios.html

[3] http://www.threedee.com/jcm/psystem/

[4] http://www.moorecad.com/standardpascal/pug_newsletter_17.pdf

[5] http://miller.emu.id.au/pmiller/ucsd−psystem−um/reconstruct/00−frontice.html

[6] http://ucsdmag.ucsd.edu/magazine/vol1no3/features/pascal.htm

[7] http://williambader.com/museum/at/psysversions.html

[8] http://www.kernelthread.com/publications/appleoshistory/1.html

[9] http://standardpascal.org/pug_newsletter_04.pdf

[10] http://standardpascal.org/pug_newsletter_08.pdf

[11] http://standardpascal.org/pug_newsletter_09_10.pdf

[12] http://standardpascal.org/pug_newsletter_11.pdf

[13] http://standardpascal.org/pug_newsletter_15.pdf

[14] http://computer−refuge.org/bitsavers/pdf/westernDigital/WD90_Pascal_Microengine/UCSD_PascalNewsl%234_Jul79.pdf

[15] http://www.museumstuff.com/learn/topics/UCSD_p-System

[16] http://en.wikipedia.org/wiki/Ucsd_pascal

BOOTSTRAPPING

This section covers how to build a UCSD Pascal system, when all you have is the source code, and none of the authors are around to ask.

The process breaks into several stages.

- The first stage is to cross compiler the minimum set of this to grep a system going.
- The second stage is to run the cross compiled system, and use it to “natively” compiler itself.
- The third stage is a double check: use the result of stage 2 to compile everything again. The result should be identical to the result of the second stage.

Stage Zero

In 2006 the UCSD issued a non-commercial royalty-free License for source code written before 1-Jun-1979 by UCSD employees. You need to download the system sources.

http://www.bitsavers.org/bits/UCSD_Pascal/ucsd_II.0/

You need a p-code interpreter (also know as a p-machine or a virtual machine) in order to run the code. Fortunately there are a number of these on the Internet, mostly written in portable C.

http://ucsd−psystem−vm.sourceforge.net/

Stage One

Compiling the source code is a challenge, if you want to completely avoid any suggestion of a proprietary taint. Usually, this means not wanting to use the proprietary Apple Pascal disk images that are available on the Internet.

You need to cross compile the system. This is needed in order to be able to run the cross compiled compiler (see next step) “natively”. This becomes the SYSTEM.PASCAL file.

You need to cross compile the native compiler. However this is not sufficient, because the native compiler uses several intrinsic units. Once compiled, this becomes the SYSTEM.COMPILER file.

You need to cross compile the PASCALIO unit, to enable the compiler to deal with real numbers. This unit is added to the SYSTEM.LIBRARY file.

You need to cross compile the LONGINTEGERS unit, to enable the compiler to deal with long integers. This isn’t complete, yet, because you must also cross assemble the DECOPS code, and link it into the long integers unit. This unit is added to the SYSTEM.LIBRARY file.

You need to cross compile the linker. This is needed so that assembler output and compiler output can be linked, in order to build completely linked units. This becomes the SYSTEM.LINKER file.

You need to cross compile the librarian. This becomes the LIBRARIAN.CODE file. This is needed to build the SYSTEM.LIBRARY file, containing all of the intrinsic units, when you perform the native build.

You need to cross compile the assembler. This becomes the SYSTEM.ASSMBLER file. This is needed to build the long integer intrinsic unit, when you perform the native build.

Use *ucsdpsys_mkfs*(1) to create a new disk image. Use *ucsdpsys_disk*(1) to copy all the system files into the disk images. You now have a minimal running system.

Stage Two

Run the system produced in Stage One, using this system repeat of Stage One, but performing native compiles, *etc*, to produce the new disk image. You now have a self-hosting system, probably.

Stage Three

This is a repeat of Stage Two, using the system produced in Stage Two. The end result should be a disk image containing files identical to that of stage two. If there are differences, there is a bug somewhere. If there are no significant differences, you now have a self-hosting UCSD p-System, for the first time singe the mid-1980s.

LESSONS

This section covers some of the things learnt while developing the cross compiler.

Motivation

The ucsd-psystem-xc project was started for two reasons. The first was a desire to try out some ideas about factory methods and how they apply to language compilers. The second was a certain nostalgia about the Apple Pascal system (base on II.1) from the early 1980s. The two just happened at the same time, and on 2006-May-22 the ucsd-psystem-xc project was the result.

Another significant 2006 event, unknown to the author at the time, was the publication by UCSD of a royalty-free License for non-commercial use of UCSD Pascal. Not an open source license, but good enough to work with.

Of course, a nagging memory of perceived limitations of the UCSD system, and its compiler, was motivation to use the skills learnt in the intervening decades to do it “better” this time around.

The early stages of the compiler resulted in an LCA’07 conference paper (available on the project web site). However the project languished after that, for lack of time. Also, the techniques had been proven, but a substantial slab of time (months, full time) would be necessary to fill in all the missing pieces.

In 2010 the opportunity came that enabled work on the code to resume. More and more UCSD Pascal system sources were becoming available on the Internet, making the idea of a self-hosted II.1 system, built from the sources, from scratch, became a realistic goal. In 2010 most of the UCSD systems running on

emulators used either Apple Pascal disk images (obviously proprietary), or other proprietary forms, and none of which had source code available.

2010-March

A system with no proprietary pieces was possible, even with the non-commercial rider. Perhaps time, and the existence of an otherwise open source implementation, will give UCSD Legal Department the needed nudge to issue an open source license, at some time in the future.

Looking at the II.0 sources, it would appear that the first block of code files is slightly different than that of Apple Pascal, the compiler I am using to verify the output of the cross compiler. This is because intrinsic units, and the ability to have more than 16 segments, were both II.1 features.

0.2, 2010-Apr-01

Version 0.2 of the cross complier was released.

Shortly after this (2010-Apr-06) the ucsd-system-vm project was initiated, as a friendly fork of Mario Klebsch's excellent p-interp project. This would permit adding features to the interpreter, and bug fixes if any were needed.

This was the beginning of my search for a p-machine test suite, on the assumption that as the UCSD team ported their system to ever more microcomputers, they would have needed a way to validate the different ports.

The built-in EXIT function

Pascal has the ugly feature

```
exit(functionName)
```

which is approximately the equivalent of a `return` statement. But is can also act as a non-local return statement. Non-local *return*? Who thought that was a good idea? An exception-lite concept, maybe? Great, but for the fact that exceptions are ugly as well.

Easy enough to implement, function names are already first-class expressions, so we can use the regular expression grammar without trouble, and adding another built-in "function".

But it gets worse. You may recall that programs are declared with a name, as in

```
program sewer;
begin
end.
```

so you can also say

```
exit(sewer)
```

and leave the program, um, "cleanly". Well, the symbol *is* in the symbol table, so we just add another type to the list of acceptable types you can use for parameter one of the exit "function".

But wait, there's more. You can also say

```
exit(program)
```

this lets you non-local-return from the program from within units, when you don't actually know the name of the program.

This requires adding another expression production to the grammar. It is implemented using a usually-inaccessible "program" symbol, that has the appropriate segment and procedure number.

The `chr` Function

The `chr` function on UCSD native compiler did unexpected things (or, unexpected by today's standards) mostly because it leaves the value on the top of the stack unchanged. This is probably not ISO 7185 conforming. For now, the cross compiler works the same way.

In the future, masking the value with 0xFF would be better, provided we optimize the mask away for STB expressions and relevant STP expressions.

Set Opcodes Asymmetric

The native compiler understands set=set, set<>set, set<=set (known as an improper subset), set>=set (improper superset), but does not implement set<set (known as a proper subset) and set>set (proper superset).

This stems from the p-machine definition, that only defines these four. But why? Adding run-time support for these would be less than 40 bytes in the interpreter, less than 20 on some architectures. Why leave them out? It is asymmetric, for very little savings.

This missing set comparisons were added to the ucsd-psystem-vm p-machine 2010-May-11, and first released 2010-May-17. The tests use the cross compiler, of course, to generate the new opcodes.

String Handling

The builtin `concat` function is sued to join strings together. It is implemented as several calls to the system `sconcat` procedure.

```
procedure sconcat(var src, dest: string; destleng: integer);
begin
  if length(src) + length(dest) <= destleng then
  begin
    moveleft(src[1], dest[length(dest) + 1], length(src));
    dest[0] := chr(length(src) + length(dest))
  end
end
```

What happens when the results don't fit? Perhaps a runtime error would be more appropriate?

This is compounded by the fact that there could be more than one string being joined together. It is implemented by the compiler as multiple calls to segment 0 `SCONCAT` procedure, one for each argument. The ugly part is that `concat(a, b, c)` will return `concat(a, c)` if `concat(a, b)` would overflow but `concat(a, c)` would not. This is because the compiler generates these calls:

```
temp := ""
sconcat(temp, a, sizeof(temp));
sconcat(temp, b, sizeof(temp));
sconcat(temp, c, sizeof(temp));
```

at which point the above `SCONCAT` code silently doing nothing on overflow looks very, very wrong.

Standards

The UCSD pascal implementation predates the efforts to standardize the Pascal language, and had many influences on the final result. Many UCSD coders were part of that process.

The problem with pre-dating the standard is that, inevitably, UCSD Pascal was not, and is not, standard conforming. There is the possibility, initially with the cross compiler, to retro-fit compliance; or, as much compliance as possible without breaking existing programs.

0.2, 2010-Apr-19

Version 0.2 of the cross compiler is released.

It is now able to compile the UCSD II.0 compiler source code. Doesn't produce a usable native compiler at this stage.

0.3, 2010-Apr-27

Version 0.3 of the cross compiler is released.

It is now able to compile all of the II.0 system code (not the whole thing, just the "system" runtime support). Doesn't produce a usable runtime at this stage.

The test p-machine test suite was found in the II.0 sources. I had downloaded the II.0 sources months earlier, and had overlooked the "diagnostics" volume.

0.4, 2010-May-06

Version 0.4 of the cross compiler is released. As of this release, all of the II.0 non-unit sources are able to be compiled. The cross compiler now has support for all built-in functions, except `str` that needs long

integer support.

When used to produce a disk image, the emulator is able to run the system, and display the system prompt, and Filer also appears to work. No attempt to use the native compiler.

The `odd` Function

The `odd` function, like the `chr` function, does not change the value on the top of the stack. This was used in some very ugly hacks to gain access to and, or and not bit-wise arithmetic:

```
a,b,c: integer;

a := ord(odd(b) and odd(c));
a := ord(odd(b) or odd(c));
a := ord(not odd(b));
```

The cross compiler does not support this usage, but instead overloads the AND, OR and NOT operators to provide hack-free access to the bit-wise opcodes of the p-machine.

The UCSD authors were comfortable extending Pascal in novel directions, so why didn't they just overload the AND, OR and NOT expressions to handle integers as well? That would result in much cleaner code than the ord/odd hack.

0.5, 2010-May-17

Version 0.5 of the cross compiler is released.

The milestone for this release was to pass II.0 the p-machine diagnostics. There were quite a few test failures to work through. This found a number of errors in the cross compiler (fixed in this release), and also a number of errors in the p-machine (also fixed).

Version 0.9 of ucsd-psystem-vm was released the same day.

Units

Getting ordinary units to work was relatively simple. The "internal" units (for want of a better term) are more difficult because they are not documented anywhere.

```
(*$U-,R-*)
program pascalsystem;

    globals...

    unit pascalio;
    interface
        unit interface...
    implementation
        unit member procedures...

    (* no BEGIN here *)
    end;

begin
end.
```

This turned out to be relatively simple, by re-using most of the stand-alone unit grammar again, deeper within the regular program grammar. Oh, and the `(*$R-*)` isn't optional.

Long Integers

Long integer support is mostly finished. It turns out that long integers can't do all the things that integers can do. In particular, the following operations are not supported: `mod`, `odd`, `abs`, and `sqr`.

While it would be possible to generate code to calculate the `mod` value (using a division, a multiply and a subtraction; *i.e.* slowly) it is frustrating that the long integer `div` already calculates the remainder, only to discard it. Implementing long integer `mod` would only be only a few extra bytes of code in DECOPS.

Another puzzle: why one DECOPS procedure, instead of 11? (Or 17, if you include the relational operators). The DECOPS selector only uses one byte per long integer operation (two bytes for comparisons), but in any non-trivial program using long integers, it is all going to add up to a non trivial overhead... even after one takes into account the additional procedure attribute and procedure dictionary penalties. And no dispatch tables or indirect jumps within DECOPS would mean faster performance, too. It's not as if that segment's procedure dictionary is full or anything. Strange.

The call interface of the DECOPS procedure means that it can't be implemented in Pascal, and then later optimized by using assembly code. The assembly DECOPS could have then been cross checked with the Pascal to verify correctness. This seems strange, given that long integer arithmetic is non-trivial to get right.

The UCSD test suites I have found to date do not include long integers.

Long Integer Implementation

Long integers are implemented as a sign word, plus ((n+3)/4) words of 4 BCD digits each. Basically unsigned arithmetic decorated with a sign.

The digits appear to have been stored in an order that made it impossible to use the 6502's native BCD support... of course the 6502 had bugs when you tried to mix BCD with interrupts, so maybe no big loss. On the other hand, the Z80 had non-modal native BCD opcodes, but the BCD digits are in the wrong order for the Z80 as well. No idea if the PDP-11 had native BCD opcodes or not.

I shudder when I recall how we were so hung up on printing numbers (the least common arithmetic activity) that we consciously rejected obviously more efficient implementations like using radix 256 or radix 64k representations.

0.6, 2010-May-30

Version 0.6 of the cross compiler is released.

The target for this release was to be able to compile units. Both ordinary units and "internal" units (for want of a better term) are able to be compiled. This is essentially II.1 functionality, rather than II.0, in that the `separate` keyword is not particularly meaningful.

The SYSTEM.LIBRARY file can't actually be completed because assembler support is required.

Ord and Odd revisited

It turns out the assembler sources use the `ord(odd(x) and odd(y))` hack when calculating hashes for its symbol table. Surely it would have been easier to overload AND, OR and NOT to accept integer parameters as well? Looking at the 2.0 native compiler sources, it only adds three lines of code.

0.7, 2010-Jun-21

This release is the first to be able to build non-trivial amounts of the ucsd-psystem-os project's source base. To facilitate this, many command now accept a "−−arch" option, to set the architecture; this makes getting the endian-ness correct much easier, and the same option is used for all ucsd-psystem-xc commands that need to know the byte sex.

The *ucsdpsys_mkfs*(1) command from the ucsd-psystem-fs project also accepts the same option, for the same reason. The ucsd-psystem-fs 1.14 release was the same day.

The assembler parts of the system library can not yet be assembled, but the sources when cross compiler produce a working system. The ucsd-psystem-os 1.1 release was the same day.

Linker Adventures

The II.0 codefile format is unchanged from the I.5 codefile format. This means that the II.0 sources from BitSavers contain the I.5f linker sources.

However, I have been using the Apple Pascal compiler, assembler, and linker to check my results against, and it is based on II.1, and the II.1 codefile format has more information in it than the II.0 (I.5) codefile format. Initially I thought this would mean that the ucsd-psystem-os project would not be able to build stage 2. However, this probably won't be a problem, because while I recall a system with intrinsic units, the II.0 sources do not contain any.

This doesn’t preclude adding intrinsic units to stage 1, and so I can proceed to implement drop-in replacements for the Apple intrinsics, if that is how things turn out.

Name Expression Factories

The code uses factories in a number of places, but not for building name expressions, for variables and the like. However, the code has been refactored so that symbols create their own name expressions. This simplifies the code, removes a bunch of down-casts, and better suits the philosophy of the rest of the code.

This permits several derivations of the symbol variable class, including global, local, unit (needing relocation), and external segments.

But how do the specialized symbols get created? By symbol factories of the scope, of course. Each scope knows what kinds of symbols it can create.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_libmap − print map of UCSD p-System code file

SYNOPSIS

ucsdpsys_libmap [*option...*] *code-file-name...*
ucsdpsys_libmap −V

DESCRIPTION

The *ucsdpsys_libmap* program is used to print out a map of a UCSD p-System code file. This is equivalent to the `LIBMAP` program which comes with the UCSD p-System, but you can run it from Unix.

OPTIONS

The following options are understood:

−d

−−debug
Increase the debug output level.

−o *filename*

−−output=*filename*
Write the output to the named file, rather than the standard output.

−P *release-name*

−−p−machine=*release-name*
This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to “II.1” if not set.

−V

−−version
Print the version of the *ucsdpsys_libmap* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_libmap* command will exit with a status of 1 on any error. The *ucsdpsys_libmap* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_disassemble(1)
disassemble a UCSD p-System code file

ucsdpsys_link(1)
UCSD p-System codefile linker

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_librarian − UCSD p-System codefile librarian

SYNOPSIS

ucsdpsys_librarian −−file *codefile* [*option*...]
ucsdpsys_librarian −−version

DESCRIPTION

The *ucsdpsys_librarian* program is used to manipulate UCSD p-System Codefiles. It has functionality similar to the LIBRARIAN program distributed with the UCSD p-System.

OPTIONS

The following options are understood:

−a *filename*

−−copy=*filename*
This option is used to name an alternate codefile from which to copy segments, using the **−−segment** option. This file must already exist, and must be a valid codefile. This option must appear on the command line *after* the **−−file** or **−−create** options. This option may be used more than once.

−c *filename*

−−create=*filename*
This option is used to name a codefile to be created. It will replace any file of the same name. This codefile will initially be empty. If you use this option, you cannot also use the **−−file** option.

−f *filename*

−−file=*filename*
This option is used to specify the codefile being manipulated. This file must already exist, and must be a valid codefile. If you use this option, you cannot also use the **−−create** option.

−l

−−list This option may be used to obtain a library listing. If use by itself, the **−−file** codefile is unchanged. If use on the end of a more complex command line, it will show you the map of the codefile it is going to write. This option must appear on the command line *after* the **−−file** option.

−n *text*

−−notice=*text*

−−copyright=*text*
This option may be used to change the text of the copyright notice embedded in the **−−file** codefile, or to add a notice to a codefile that does not already have one. You will probably need to use quotes to insulate white space and punctuation characters from the shell (or you can use underscores instead of spaces, and they will be replaced by spaces). To remove the copyright notice, use the empty string; you will need to quote it. This option must appear on the command line *after* the **−−file** option. The notice is limited to 79 characters, more than that will be silently truncated.

−o *filename*

−−output=*filename*
Usually the *ucsdpsys_librarian* command makes the changes to the **−−file** codefile in-place. This option is used to select a different codefile to receive the results. This option may not be used with the **−−create** option.

−P *release-name*

−−p−machine=*release-name*
: This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to “II.1” if not set.

−r *name*

−−remove=*name*
: This option is used to remove the named segment from the **−−file** codefile. It is an error if the segment is not present. The segment name is *not* case sensitive. This option must appear on the command line *after* the **−−file** option. This option may be used more than once.

−R *name*

−−force-remove=*name*
: This option is used to remove the named segment from the **−−file** codefile. It is **not** an error if the segment is not present. The segment name is *not* case sensitive. This optional also works for segment numbers. This option must appear on the command line *after* the **−−file** option. This option may be used more than once.

−X

−−remove-system-segments
: This option is used to remove segments 0 and 2..6 from a codefile. This can be necessary when a `(*$U−*)` utility program contains dummy system segments.

−s *name*

−−segment=*name*
: The option is used to a name segment from the **−−copy** codefile to be added to the **−−file** codefile. The segment name is *not* case sensitive. This option must appear on the command line *after* the **−−copy** option. This option may be used more than once.

  If the form **−−segment** *name*=*number* is used, the segment will also be renumbered to the given segment number.

−V

−−version
: Print the version of the *ucsdpsys_librarian* program being executed.

All other options will produce a diagnostic error.

EXAMPLES

This section contains a few example commands.

List the segments in the file

You can obtain a list of the segments in the codefile using the following command.

```
ucsdpsys_librarian −−file example.code −−list
```

This will produce the same output as the *ucsdpsys_libmap*(1) command.

Copyright Notice

You can change the copyright notice in a codefile using the following command.

```
ucsdpsys_librarian −−file example.code \
    −−notice "Copyright (C) 1812 Tchaikovsky"
```

The same command can be used to add a copyright notice to a codefile that doesn’t have one.

Remove Segments

You can remove a segment from a codefile using the following command.

```
ucsdpsys_librarian −−file example.code −−remove DUMMYSEG
```

The segment name is *not* case sensitive.

Transfer Segments

You can transfer segments between codefiles using the following command.

```
ucsdpsys_librarian --file example.code \
    --copy fromhere.code --segment EXAMPLE
```

The segment name is case **in**sensitive.

In this example, all of the segments in “example.code” are preserved. If there was already a segment called EXAMPLE it will be replaced. The “fromhere.code” codefile will be unchanged.

New Library

You can create a new library form a series of other codefiles using a command such as

```
ucsdpsys_librarian --create=system.library \
    --copy pascalio.code --segment pascalio=31 \
    --copy long_integer.code --segment longinte=30 \
    --copy transcendental.code --segment transcen=29
```

This leaves the contributing codefiles unchanged, and creates a completely new file file to hold the results.

EXIT STATUS

The *ucsdpsys_librarian* command will exit with a status of 1 on any error. The *ucsdpsys_librarian* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_assemble(1)
: UCSD p-System cross assembler, for multiple CPU types

ucsdpsys_compile(1)
: A cross compiler from Pascal to UCSD p-System codefiles.

ucsdpsys_disassemble(1)
: A utility to disassemble UCSD p-System codefiles.

ucsdpsys_libmap(1)
: Print segment maps of UCSD p-System codefiles.

ucsdpsys_link(1)
: UCSD p-System codefile linker

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_link − UCSD p-System codefile linker

SYNOPSIS

ucsdpsys_link [*option*...] *filename*...

ucsdpsys_link −−version

DESCRIPTION

The *ucsdpsys_link* program is used to link incomplete programs with the library units and external procedures and functions that complete them.

OPTIONS

The following options are understood:

−d

−−debug
: This option may be used to increase the debug output. The more time given, the more voluminous the output.

−m *filename*

−−map=*filename*
: This option may be used to request that a link map be written to the given file.

−n *text*

−−notice=*text*

−−copyright=*text*
: This option may be used to set the copyright notice of the output codefile.

−o *filename*

−−output=*filename*
: The option is used to specify the output file. Not optional.

−P *release-name*

−−p−machine=*release-name*
: This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to “II.1” if not set.

−v

−−verbose
: This option may be used to request progress information during the link. This was useful in 1.0MHz days, not so useful now.

−V

−−version
: Print the version of the *ucsdpsys_link* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_link* command will exit with a status of 1 on any error. The *ucsdpsys_link* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_assemble(1)
: UCSD p-System cross assembler, for multiple CPU types

ucsdpsys_compile(1)
: A cross compiler from Pascal to UCSD p-System codefiles.

ucsdpsys_disassemble(1)
: A utility to disassemble UCSD p-System codefiles.

ucsdpsys_librarian(1)
: UCSD p-System codefile librarian

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_littoral − read UCSD Pascal and write C++

SYNOPSIS

ucsdpsys_littoral [*option*...] *filename*...

ucsdpsys_littoral −−version

DESCRIPTION

The *ucsdpsys_littoral* program is used to read a UCSD pascal program and write something that is nearly, almost, but not quite, C++. This can very helpful when trying to replicate the functionality of a UCSD Pascal program in a more modern environment.

OPTIONS

The following options are understood:

−o *filename*

−−output=*filename*

This option may be used to select where the output is written. It defaults to the name of the source `.text` file, with the extension removed and "`.cc`" appended.

−V

−−version

Print the version of the *ucsdpsys_littoral* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_littoral* command will exit with a status of 1 on any error. The *ucsdpsys_littoral* command will only exit with a status of 0 if there are no errors.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_opcodes − UCSD p-System system.opcodes generator

SYNOPSIS

ucsdpsys_opcodes −e *text-file binary-file*
ucsdpsys_opcodes −d *binary-file text-file*
ucsdpsys_opcodes −V

DESCRIPTION

The *ucsdpsys_opcodes* program is used to read a text template of an assembler's opcode file and write the equivalent binary file. This is used by the *ucsd-psystem-os* project, when building the assembler and disassembler programs.

OPTIONS

The following options are understood:

−A *name*

−−architecture=*name*
: This option is used to indicate the translation to be performed, and also the byte sex of the binary files. The architecture name can usually be found in the name of the binary *name*`.OPCODES` binary files. If a p-code mtype is given, it indicates that the data file is to be used by the p-code disassembler, the `OPCODES.II.0` file.

−e

−−encode
: Encode a text file into a binary file. This is used to have an editable text representation of the file contents, so that they can easily be edited, and version controlled.

−d

−−decode
: Decode a binary file into a text file. This can be used to reverse-engineer the text file from the existing binary files. With the passing of time, the method originally used to create the binary files has been lost.

−V

−−version
: Print the version of the *ucsdpsys_opcodes* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_opcodes* command will exit with a status of 1 on any error. The *ucsdpsys_opcodes* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_assemble(1)
: UCSD p-System cross assembler, for multiple CPU types

ucsdpsys_opcodes(5)
: format of the OPCODES.II.0 file

COPYRIGHT

ucsdpsys_opcodes version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_osmakgen − write the Makefile for the ucsd-psystem-os project

SYNOPSIS

ucsdpsys_osmakgen [*option*...][*filename*...]
ucsdpsys_osmakgen −−version

DESCRIPTION

The *ucsdpsys_osmakgen* program is used to write the `Makefile` file for the ucsd-psystem-os project, based on the file names provided.

The generated `Makefile` file uses the *ucsdpsys_compile*(1) cross compiler to bootstrap a UCSD p-System from sources alone. The executables are then combined into disk images, using the *ucsdpsys_mkfs*(1) and *ucsdpsys_disk*(1) file system tools. The *ucsdpsys_assemble*(1) multi-target cross assembler is used to assemble any necessary assembler code. The *ucsdpsys_setup*(1) tool is used to translate a text representation of SYSTEM.MISCINFO, if present, into the binary form.

The source files will be scanned, using *ucsdpsys_depends*(1), for include dependencies, and the results incorportaed into the generated `Makefile` file.

Directory Structure

The *ucsdpsys_osmakgen* command takes its cues from the names of the files you give to it.

program`/main.text`
: The existence of a file with this name pattern indicates that the *program* is to be compiled using a Pascal compiler. That file can always include other files, but only from the same directory. The codefile resulting from the compilation will be placed into `stage`*n*`/`*host*`/codefiles/`*program*`/main.code`

  If there is an assembler component, the output of the compilation is `main.pas.code`, and then the *ucsdpsys_link*(1) command is used to link `main.pas.code` and `main.asm.code` to form the final `main.code` file.

`arch/`
: This directory contains subdirectories, each one named for a specfic microprocessor. Each of these subdirectories contain various programs and library components specific to that microprocessor. Examples include “pdp11”, “z80”, and “6502”.

`arch/$(arch)/`
: Within the generated `Makefile` file, the relevant subdirectoiry is always accessed using this construct.

`arch/$(arch)/`*program*`/main.asm.text`
: The existence of a file with this name pattern indicates that the *program* is to be built using a cross assembler. That file can always include other files, but only from the same directory. The codefile resulting from the assembly will be placed into `stage`*n*`/`*host*`/codefiles/`*program*`/main.asm.code`

  The *ucsdpsys_link*(1) command is used to link `main.pas.code` and `main.asm.code` to form the final `main.code` file.

  Note: the assembler component is in a directory named the same as the Pascal portion of the program. However, it is an *arch* sub-directory, so that it is possible to have all of the sourec code, including all architecture variants, in the same source tree.

`arch/$(arch)/assembler/main.text`
: Each microprocessor as its own assembler, and this is where it may be found. Note we have not yet recovered the source code to all of the assemblers, so coverage will be inconsistent.

`arch/$(arch)/assembler/error-messages.text`
: This file, if present, will be used to create the `$(arch).ERRORS` data file, by processing it with the *ucsdpsys_errors*(1) command.

`arch/$(arch)/assembler/opcode-data.text`
: This file, if present, will be used to create the `$(arch).OPCODES` data file, by processing it with the *ucsdpsys_opcodes*(1) command.

`host/` This directory contains subdirectories, each one named for a specfic host system (brand name) that are implemented using a micro processor, but include varying sets of peripherals. Each of these subdirectories contain various programs and library components specific to that host hardware. Examples include “terak”, “cpm”, and “apple”.

`host/$(host)/`
: Within the generated `Makefile` file, the relevant subdirectoiry is always accessed using this construct.

`host/$(host)/miscinfo.text`
: This is the location of the host-specific text source file of the `SYSTEM.MISCINFO` data file.

`stage`*n*`/system.syntax`
: This file is constructed from the `compiler/error-messages.text` file, if present.

OPTIONS

The following options are understood:

−A *name*

−−architecture=*name*
: This option may be used to specify an alternative architecture name in the generated Makefile. By default, it is calculated from the name of the host. You should need this option very rarely.

−b

−−no-blurb
: This option may be used to suppress the extensive comments generated into the Makefile.

−c *number*

−−change=*number*
: This option is used to specify the number of the Aegis change set to ask for the list of file names.

−C *text*

−−copyright=*text*
: This option may be used to specify the copyright notice to be attached to the system library. Use the empty string to have no copyright notice.

−H *name*

−−host=*name*
: This option may be used to specify an alternative host system name. Defaults to “klebsch”, in reference to the *ucsdpsys_vm*(1) interpreter written by Mario Klebsch.

−o *filename*

−−output=*filename*
: This option may be used to specify the name of the file to be written. Defaults to “Makefile” if not given. The name “−” is understood to mean the standard output.

−p *name*

−−project=*name*
: This option is used to specify the name of the Aegis project to ask for the list of file names. If you specify neither an Aegis change set nor an Aegis project name, only filename named omn the command line are considered.

−V

−−version
Print the version of the *ucsdpsys_osmakgen* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_osmakgen* command will exit with a status of 1 on any error. The *ucsdpsys_osmakgen* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_assemble(1)
UCSD p-System multi-target Cross Assembler

ucsdpsys_compile(1)
UCSD p-System Cross Compiler

ucsdpsys_depends(1)
Include file dependency finder.

ucsdpsys_disk(1)
UCSD p-System disk image manipulation.

ucsdpsys_mkfs(1)
UCSD p-System disk image creator.

ucsdpsys_setup(1)
UCSD p-System SYSTEM.MISCINFO encoder and decoder

ucsdpsys_vm(1)
UCSD p-System virtual machine (p-code interpreter).

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_pretty − UCSD p-System Pascal pretty printer

SYNOPSIS

ucsdpsys_pretty [*option...*] *filename*
ucsdpsys_pretty −−version

DESCRIPTION

The *ucsdpsys_pretty* program is used to re-format the source file of a Pascal program. The re-formatted program is written to the standard output.

OPTIONS

The following options are understood:

−d

−−debug
: This option may be used to increase the verbosity of debug output. May be specified more than once.

−f*name*

−−feature *name*
: This option may be used to turn enable or disable the various features. See *ucsdpsys_compile*(1) for more information.

−I*directory*

−−include *directory*
: This option may be used to nominate a directory to be search for include files. This option may be used more than once.

−o*filename*

−−output *filename*
: This option may be used to redirect the output to the named file. If not specified, output will be written to the standard output.

−V

−−version
: Print the version of the *ucsdpsys_pretty* program being executed.

−W*name*

−−warning *name*
: This option may be used to enable or disable the various warnings. See *ucsdpsys_compile*(1) for more information.

−y

−−grammar−trace
: Turn on parse debugging. Very verbose. Intended for compiler developers only.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_pretty* command will exit with a status of 1 on any error. The *ucsdpsys_pretty* command will only exit with a status of 0 if there are no errors.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_setup − manipulate the SYSTEM.MISCINFO file

SYNOPSIS

ucsdpsys_setup −e *textfile datafile*

ucsdpsys_setup −d *datafile textfile*

ucsdpsys_setup −V

DESCRIPTION

The *ucsdpsys_setup* program is used to encode and decode the `SYSTEM.MISCINFO` configuration file.

This configuration file is read by `SYSTEM.PASCAL` when the system boots, and is used to control how the system output terminal is used for many operations. It is also used by editors (`SYSTEM.EDITOR` or `YALOE`) to understand how to manipulate the screen.

Think of it as a very poor *terminfo*(3) substitute. The *ucsdpsys_vm*(1) virtual machine takes care of translating the terminal control characters, and uses *curses*(3) to be terminal independent.

The result of the *ucsdpsys_setup −d* command is in the same format as the *ucsdpsys_setup −e* expects as input. This means you can, for example, track its contents with a version control system.

Field Values

A effort has been made to be as similar as possible to the original `SETUP` program, except that it isn't interactive. The fields all have the same names as the original `SETUP` program, and accept similar values.

boolean The values `true` and `false`, and several synonyms for each, are understood.

char The values are given as decimal integers, or control characters may be given using their tradition three-character ASCI names.

integer Integer values fields may be given as decimal text.

OPTIONS

The following options are understood:

−A *name*

−−architecture=*name*
: This option may be used to specify the machine type this file describes. This is used to encode and decode the 16-bit fields in the data.

−d

−−decode
: This option is used to decode the binary form into the text form.

−e

−−encode
: This option is used to encode the text form into the binary form.

−P *release-name*

−−p−machine=*release-name*
: This option may be used to select the p-machine of interest. This has to do with the shape of codefiles (segment dictionaries, and the available opcodes). This defaults to "II.1" if not set.

−V

−−version
: Print the version of the *ucsdpsys_setup* program being executed.

All other options will produce a diagnostic error.

EXIT STATUS

The *ucsdpsys_setup* command will exit with a status of 1 on any error. The *ucsdpsys_setup* command will only exit with a status of 0 if there are no errors.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program − to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product,

and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing

the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to

infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF

SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

> *one line to give the program’s name and a brief idea of what it does.*
> Copyright (C) *year name of author*
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

> <program> Copyright (C) <year> <name of author>
> This program comes with ABSOLUTELY NO WARRANTY; for details type “show w”. This is free software, and you are welcome to redistribute it under certain conditions; type “show c” for details.

The hypothetical commands “show w” and “show c” should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

NAME

ucsdpsys_codefile − UCSD p-System codefile format

DESCRIPTION

The UCSD p-System codefiles serve as both intermediate object files (the equivalent of Unix `.o` files), and executables (the equivalent of Windows `.exe` files).

By convention, UCSD codefiles have a `.code` suffix, and their file type in the filesystem is set code `CODE` as well. For most commands that deal with codefiles, the UCSD system will automatically append `.code` if the user does not.

As the UCSD p-System evolved, so did the codefile format. Most of these changes are backwards compatible (later versions of the p-System can read the codefiles of previous versions), and a few are forwards compaitible (earlier versions of the p-System can cope with codefiles from later versions of the system).

A word of caution: while the I.5 linker can mostly handle II.0 codefiles, the p-code is different between the two systems. Do not try to run a <= I.5 executable on a >= II.0 system, or *vice versa*.

By examining the Segment Dictionary, it is usually possible to determine whether a codefile is I.5 or earlier, or II.0 or later.

CODEFILE LAYOUT

A UCSD p-System codefile has a one-block segment dictionary, followed by each of the segments.

Segment Dictionary
First Segment
Second Segment
etc

Note that segments do not have to appear in order, although they frequently do. The Segment Dictionary is able to place segments anywhere in the codefile.

Each segment has three parts:

Interface Text (may be empty)
Procedures
Link Info (may be empty)

While the Segment Dictionary format (see below) would suggest that Interface text can be placed anywhere in the codefile, rather than immediately before the Procedure code, in practive this is not done, thus providing an indirect way of knowing how many blocks of interface text are present.

SEGMENT DICTIONARY

The Segment Dictionary contains the locations of the segments, and also some meta-data about each segment. Historically, it has been added to with each major release.

Byte Sex

The UCSD p-System can be hosted by both little-endian and big-endian machines. The native byte ordering of the host is used in codefiles, including the segment dictionary. Whenever you see integers (16-bits) and packed records, byte sex must be taken into account.

Version I.5

The Pascal declaration looks like this:

```
record
  diskinfo: array [0..15] of
    record
```

```
            codeaddr: integer;
            codeleng: integer
          end;
        segname: array [0..15] of array[0..7] of char;
        segkind: array [0..15] of integer;
        textaddr: array [0..15] of integer;
        filler: array [0..87] of integer;
        comment: string[79]
      end;
```

The record field are defines as follows:

codeaddr
: The start of the segment's code, in units of 512-byte blocks.

codeleng
: The sizeof of the segment's code, in bytes.

segname
: The segment's name, truncated to 8 bytes, padded with spaces on the right if necessary. Must be upper case. Must consist of letters and digits only, as this is checked by the UCSD p-System native linker.

segkind
: The segment kind.

  LINKED (0)
  : No work is needed for this segment, it is executable as is.

  HOSTSEG (1)
  : PASCAL host program outer block, when there is at least one EXTERNAL procedure or function. Not executable.

  SEGPROC (2)
  : PASCAL segment procedure, not host

  UNITSEG (3)
  : If *codeleng* is non-zero, this is a library UNIT definition. If *codeleng* is zero, this is a library UNIT reference.

  SEPRTSEG (4)
  : The assembler procuces this kind of segment. It is populated entirely with native code procedures and functions. You use the linker to link HOSTSEG segments with SEPRTSEG segments to produce LINKED segments.

textaddr
: The text of the INTERFACE section of a UNIT. Always ends with an IMPLEMENTATION keyword followed by ten (10) spaces. It is in the usual textfile format (see *ucsdpsys_text*(5) for more information) except that (a) it does not have the two block editor header, and (b) it could be an odd number of blocks long, if the last block would have been all NUL bytes.

filler
: Unused. Must be filled to zero, so that zero can be used as the defualt value for backwards compatibility.

comment
: The is the copyright string added to the codefile using the `(*$C` *comment* `*)` control comment in the source code. It will be empty if none was given; fill unused bytes with zero.

Version II.0
: The II.0 codefiles had the same format as the I.5 codefiles. The big change in II.0 were the alterations to the opcodes.

Version II.1
: The Pascal declaration looks like this:

```
      record
```

```
        diskinfo: array [0..15] of
          record
            codeaddr: integer;
            codeleng: integer
          end;
        segname: array [0..15] of array[0..7] of char;
        segkind: array [0..15] of integer;
        textaddr: array [0..15] of integer;
        seginfo: array [0..15] of
          packed record
            segnum: 0..255;
            mtype: 0..15;
            version: 0..7
          end;
        intrins: set [0..63] of boolean;
        filler: array [0..67] of integer;
        comment: string[79]
      end;
```

Most of the fields are the same as for I.5. The differences are:

segkind
: There are several new values

seginfo.segnum
: blah blah blah

seginfo.mtype
: blah blah blah

seginfo.version
: blah blah blah

Version IV

The Pascal declaration looks like this:

dictionary chaining

endian word

COPYRIGHT

ucsdpsys_codefile version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

SEGMENT LAYOUT

Each segment has three parts:

Interface Text (may be empty)
Procedures
Link Info (may be empty)

Interface Text

write this section. see above.

Procedures

The procedures section is indicated by the *codeaddr* and *codeleng* fields in the Segment Dictioary. The procedures are laid out as

First Procedure
Second Procedure
etc
Procedure Dictionary

The procedure dictionary can be located by using the *codeleng* value, because it appears and the **end** of the segment's code.

Procedures do not always appear in the segment in strict numerical order. Procedure numbers are allocated by the compiler when it sees a procedure declaration. If it is declared `forward`, or if it has nested procedures, the code of other procedures may appear before it in the segment.

etc	
Procedure 2 Pointer	
Procedure 1 Pointer	
Segment Number	Number of Procedures

The shortes segment ins two bytes: one byte for the segment number, and one byte for the procedure count (zero).

Link Information

write this section

COPYRIGHT

ucsdpsys_codefile version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_errors − UCSD p-System assembler error file format

DESCRIPTION

The UCSD Adaptive Assembler uses a binary error file. This simplifies the error processing, at the cost of

- error texts of limited length,
- significant difficulty of editing, and
- when measured in whole 512-byte blocks, little or no space savings.
- Inconsistent with the textual error file format used by the compiler.

By using a text file as the primary source, it can be edited easily, and placed under version control. The binary file can be created from the text file using the *ucsdpsys_errors*(1) command.

Format of the Text file

The text file is the same as for the compiler.

Comments have a hash (“#”) in the first column, and extend to the end of the line. Blank lines are ignored.

Each error message has a number, a colon (“:”), and the text of the error message. Excess white space is discarded. The lines do not need to be in order, but there may be no duplicates. Error text lengths in excess of 40 characters are an error.

Format of the Binary file

The error file declared as

```
type error_string: string[40];
var error_file: file of error_string;
```

That is, each error occupies 42 bytes of the file, even if the error text is significantly shorter. There is no way to cram a longer error message into the file.

The file is indexed by error number, using

```
seek(error_file, error_num);
```

Error numbers that are not used contain a value of one space. Error zero exists in the file, and is unused.

EXIT STATUS

The *ucsdpsys_errors* command will exit with a status of 1 on any error. The *ucsdpsys_errors* command will only exit with a status of 0 if there are no errors.

SEE ALSO

ucsdpsys_errors(1)
UCSD p-System assembler error file builder

COPYRIGHT

ucsdpsys_errors version 0.11

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/

NAME

ucsdpsys_opcodes − format of the OPCODES.II.0 file

DESCRIPTION

The *ucsdpsys_opcodes*(1) command is used to generate the binary *name*`.OPCODES` system file expected by the UCSD p-System assemblers. It can also be used to reverse engineer and existing file into the text equivalent.

The *ucsdpsys_opcodes*(1) command is used to generate the binary `OPCODES.II.0` system file expected by the UCSD p-System disassembler (for p-code architectures). It can also be used to reverse engineer and existing file into the text equivalent.

Format if the Assembler text file

Comments are permitted, they start at a hash (#) character, and finish at the end of the line.

Each opcode is a line of the form

```
{ "name", value, type },
```

Where *type* is one of the known opcode type (see the Adaptive Assembler sources for more information).

The resemblance to a C initializer is not a coincidence.

Format if the Disassembler text file

Comments are permitted, they start at a hash (#) character, and finish at the end of the line.

The opcode lines make take one of two forms

```
number = type, "name";
number = type;
```

The first format describes most of the lines in the file. The second format describes undefined opcodes, or opcodes with names already known to the disassembler.

Format of the Assembler binary file

Each line of the source file is encoded into 12 bytes in the binary file.

0..7 The name, space padded on the right

8, 9 The value. The byte sex depends on the architecture.

10, 11 The opcode type.

The first 12 bytes are treated differently. They indicate the byte sex of the file. All bytes are zero, except for the value bytes; they are to evaluate to 1, if you have the byte sex correct.

Format of the Disassembler binary file

There are two parts to the file: the opcode names and the opcode types.

For the opcode names, each entry in the file is 8 bytes wide, and space padded, indexed by opcode number. Absent entries are set to all spaces. The first 52 opcodes do not appear in the table.

For the opcode types, each entry in the table is 2 bytes wide.

Just why they felt the need for a file formatted differently than the assembler's data file is a mystery. The answer is lost in the mists of time.

EXIT STATUS

The *ucsdpsys_opcodes* command will exit with a status of 1 on any error. The *ucsdpsys_opcodes* command will only exit with a status of 0 if there are no errors.

COPYRIGHT

AUTHOR

Peter Miller E-Mail: pmiller@opensource.org.au
/\/* WWW: http://miller.emu.id.au/pmiller/