WERCS Language Details

This section details the language specific details of the name files produced by WERCS.

Assembly Language

If you have selected Assembler from the Language dialog box WERCS will produce a file with extension .I containing EQU statements of the form:

```
label EQU 1
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
INCLUDE TEST.I
```

The characters allowed in names are:

A-Z, a-z and _ as the first character and:

A-Z, a-z, 0-9, _ and . in subsequent characters.

Although designed with DevpacST in mind, the .I file can be used with other assemblers that follow the Motorola standard.

BASIC

If you have selected BASIC from the Language dialog box WERCS will produce a file with extension .BH containing CONST statements of the form:

```
CONST label%=1
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
rem $include test.bh
```

The characters allowed in names are:

A-Z, a-z as the first character and:

A-Z, a-z, 0-9, _ and . in subsequent characters.

The .BH file is designed with HiSoft BASIC and Power BASIC in mind, and adaptation to other BASICs for the Atari ST is straightforward straightforward so long as the BASIC is suitable for serious GEM work.

C

If you have selected C from the Language dialog box WERCS will produce a file with extension .H containing #define pre-processor statements of the form:

```
#define label 1
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
#include "TEST.H"
```

The characters allowed in names are:

A-Z, a-z and _ as the first character and:

A-Z, a-z, 0-9, and _ in subsequent characters.

All sixteen characters of the name are significant as is the case with Lattice C 5. This may cause problems with some other C implementations, such as the HiSoft C Interpreter, if your names have the first eight characters the same.

FORTRAN

If you have selected FORTRAN from the Language dialog box WERCS will produce a file with extension .INC containing PARAMETER definitions of the form:

```
INTEGER*4 LABEL
PARAMETER (LABEL=1)
```

The characters allowed in names are:

A-Z, a-z as the first character and:

A-Z, a-z ,0-9, and _ in subsequent characters.

All sixteen characters of the name are significant as in Prospero FORTRAN; the WERCS output was designed for use with this compiler and with Prospero's GEM bindings; they may not be appropriate for other FORTRANs.

We would like to apologise for the lack of an example FORTRAN program; we do not have the necessary FORTRAN expertise to produce this.

Modula-2

If you have selected Modula from the Language dialog box WERCS will produce a file with extension .DEF containing a definition module. For example, if you have saved a resource file called TEST.RSC, the file will be of the form:

```
DEFINTION MODULE TEST;
CONST label=1;
.....
END TEST.
```

If you are writing a one-module program you can use an implementation module of the same name.

Otherwise, you will need to write a implementation module like this:

```
IMPLEMENTATION MODULE TEST;
END TEST.
```

To access the constants from your other modules use:

```
FROM TEST IMPORT label;
```

or

```
IMPORT TEST;
```

and then access the labels as, for example, TEST.label.

Remember not to use the same name as your main module if you are using this method.

The characters allowed in names are:

A-Z, a-z, $ and _ as the first character and:

A-Z, a-z, 0-9, $ and _ in subsequent characters.

All sixteen characters of the name are significant.

Although designed for use with FTL Modula-2, the name files may be used with any Modula-2 compiler that follows the Third Edition of Wirth's book.

Pascal

If you have selected Pascal from the Language dialog box WERCS will produce a file with extension .INC containing constant definitions of the form:

```
CONST
  label=1;
....
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
{$I TEST.INC }
```

The characters allowed in names are:

A-Z, a-z as the first character and:

A-Z, a-z, 0-9, and _ in subsequent characters.

All sixteen characters of the name are significant.

Although our example programs are for Personal Pascal, the .INC files generated are suitable for use with most Pascal compilers.

The WTEST Example Programs

Compiling WTEST

To illustrate the most common resource-handling programming requirements, we supply an example program written in a variety of languages. The programs all do the same thing but the implementations vary according to the language used. A ready-to-run version called WTEST.PRG is supplied on the master disks; it needs WRSC.RSC to run.

To re-compile WTEST first you need to run WERCS and then use the Load command from the File menu to load WRSC.RSC from your backup disk. When WRSC.RSC is loaded the name file, WRSC.HRD containing the names of the forms and objects will be loaded as well.

All the following commands are on the File menu. Click on Language and then select the language that you wish to use by clicking on the appropriate radio button in the dialog box. Next save the file using Save; this will re-save the WRSC.RSC resource file, the name file WRSC.HRD and also a file for the language of your choice. Next use Quit to leave WERCS.

One more general point: if you edit a resource file and change its structure you will need to re-compile your program, in case any constants have changed.

C

WTEST.C is the source file for use with Lattice C 5. This is an extended version of the standard WTEST, only supplied with Lattice C 5, which rather than using square buttons, uses rounded ones. Several other headers files supplied as part of Lattice C 5 are also used.

Assembly Language

WTEST.S is the source file for use with DevpacST2. As well as the WRSC.I file produced by WERCS you will need GEMMACRO.S and AESLIB.S from your DevpacST master disk.

BASIC

WTEST.BAS is the HISoft BASIC version which will also compile under Power BASIC. It needs the GEMAES.BH file from your BASIC master disk as well as the WRSC.BH file produced by WERCS.

Modula-2

WTEST.MOD is the FTL Modula-2 version. You will need to compile the file WRSC.DEF produced by WERCS before you compile WTEST.MOD. You will also need to compile WRSC.MOD from your WERCS backup disk before you link.

Pascal

WTEST.PAS is the Personal Pascal version; to compile this you will need GEMSUBS.PAS from your Personal Pascal backup disk.

WTEST structure

The different versions of WTEST all do the same thing but the implementations vary according to the language used. We have tried to keep variable names and procedure/function names as consistent as possible.

The program is deliberately over-simplified; it manages to avoid calling the VDI completely and gets away with an evnt_mesag, avoiding the dreaded evnt_multi. The general structure of the code in all the programs is as follows:

Procedure INITIALISE

This does the required GEM initialisation then loads the resource file. The tree address of the menu is found and the menu installed. The usable screen size is found and certain global variables initialised.

Procedure SETDESK

This sets the new desktop pattern to be a particular address and forces the AES to re-draw the whole screen.

Procedure DEINITIALISE

Resets any installed desktop, removes the menu bar, frees the resource, then does any required GEM de-initialisation.

Procedure HANDLE_DIALOG

A general dialog box handler which starts by centring and drawing the box. User interaction is handled by form_do and, on return, if the exit object was a Button, it is de-selected.

Procedure SET_TEDINFO

This allows a particular TEDINFO structure to have its data portion set to a particular string.

Procedure GET_TEDINFO

Allows a particular TEDINFO structure to return its data portion.

Procedure ROUND_BUTTON

This is the code that the ProgDef in the Lattice C 5 version of WTEST uses.

Procedure OBJ_INIT

This modifies the button objects in the dialog box to be ProgDefs. The strings already there are saved and used subsequently for the round buttons. It also determines the AES's VDI handle for the use of ROUND_BUTTON. This routine is only supplied in the Lattice C 5 version.

Procedure SET_BUTTON

This allows one particular radio Button to be set from a group. If invalid parameters are specified the routine will never finish.

Procedure GET_BUTTON

Allows a group of radio Buttons to be interrogated to see which is selected.

Procedure TEST_DIALOG

This handles the particular dialog box in the resource file. It implements proper cancelling - that is, if it is cancelled, the Button state and Text entry are left alone.

Procedure HANDLE_MENU

This is the menu-click dispatcher; it takes various actions, depending on which menu item has been clicked, and also de-selects the menu title.

Procedure MAIN

This is the main loop, acting only on MN_SELECTED message events. In a proper program evnt_multi would be used and a far greater selection of cases would have to be dealt with.

HRD file format

.HRD files consist of a header record, any number of variable length data records and then an end-of-file record.

HRD Header record

Name	Size	Meaning
version	word	1 at present
autonaming	byte	1 if auto-naming selected 0 if no auto-naming
langflag	byte	1 if C, 2 if Pascal, 4 if Modula-2, 8 if FORTRAN, 16 if assembler, 32 if BASIC
autosnap	byte	0 if no character-snap 1 if half character-snap 2 if full character-snap
casing	byte	0 if mixed 1 if upper 2 if lower
autosizing	byte	1 if auto-sizing 0 if no auto-sizing
reserved	byte	not used at present

HRD Data Record

Name	Size	Meaning
type	byte	0 if Form, 1 if Menu, 2 if Alert, 3 if Free String, 4 if Free Image, 5 if object (rather than tree), 6 if end-of-file record, 7 if record names a prefix rather than a name.
reserved	byte	not used at present
treeIndex	word	number of tree
objIndex	word	if object then object number within tree
name	varies	Name terminated with a single null

LNG file format

The WERCS.LNG file is a text file containing the information that WERCS uses to work out which name file to produce, what to output and what is a valid name. You can modify this if you wish; though be warned that it is easy to produce files that your compiler won't like.

The file consists of a number of Language specifications followed by an end record. Each Language specification starts with a line like:

```
*LANGUAGE C
```

where C is the name of the language. This is used purely for documentation purposes; it won't affect the Language Dialog Box for example. The records are for C, Pascal, Modula-2, FORTRAN, assembler, and BASIC in that order.

The end record is a single line:

```
*END
```

and should be the last line in the file; the information following it will be ignored.

The other lines in the language specification may appear in any order and can be one of:

```
*SOURCE .H
```

This specifies the extension of the source file that WERCS will generate (.H in this example); the full stop (.) must be present.

```
*SIGNIFICANCE 16
```

Specifies the number of significant characters in names; minimum 1 maximum 16. The default is 16.

```
*INITIAL a-z,A-Z,_
```

Specifies the characters that are allowed as the first letter of the name. The hyphens (-) indicate a range of letters. The commas may be omitted. The equivalent of this would be

```
*INITIAL a-zA-Z_
```

which is far less obvious. Do not put spaces in the INITIAL string. The default is a-z, A-Z.

```
*FOLLOW a-Z,A-Z,0-9,_
```

specifies which characters are allowed in names other than in the first position. The syntax to specify the characters is the same as for *INITIAL. Default is a-z, A-Z, 0-9.

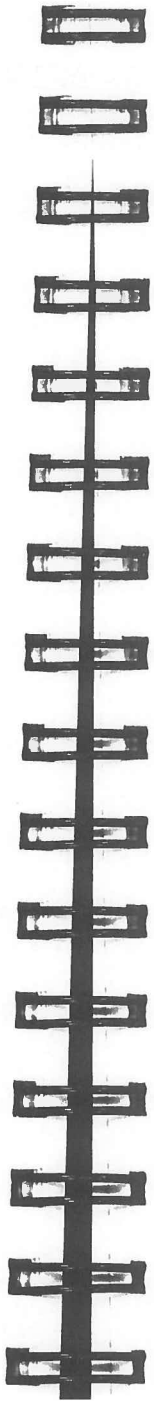
```
*INIT
*TREE
*OBJECT
*EXIT
```

These commands specify the text that is generated in the source file. The *INIT text is generated once at the top of the file, *END at the end of the file, *TREE for each named tree in the file and *OBJECT for each named object. Each entry may be more than one line long, the entry being terminated by the next * command. The following special pairs of characters may be used:

Character	Meaning
% F	base file name (without drive, directory or extension),
% T	name of the current tree,
% N	name of the current object,
% V	the value of the current object (for *OBJECT) or tree (for *TREE),
% %	a single % character.

Of the above only %F and %% should be used in *INIT and *EXIT. The default for *INIT, *EXIT, *OBJECT and *TREE is no text at all.

See the WERCS.LNG file for an example. If you want to generate code for a language that is not supported already, modify the definition of another language that you do not use. If you are adventurous you can even change the name of the Button in the Language box in WERCS.RSC but if something goes wrong don't blame us!



Appendix C Converting to Lattice C 5

Lattice 3.04

Old Lattice 3.04 programs are probably the easiest programs to convert to Lattice C 5. Often few or no changes will be required. There have however been many changes to gain ANSI conformance on the runtime libraries and some rationalisation of names to match those used by Lattice C on other architectures. The important changes for upgrade users are:

- abs has been changed so that it only deals with the type int.
- malloc, calloc, free, realloc have been extended to respond correctly to passing of ANSI NULL arguments. Note that this means that *any* use of malloc(0) will return a NULL pointer. Also note that free returns the 'type' void, not int as with the previous release.
- An extra level of data hiding has been introduced between the ANSI file and GEMDOS file handling. In particular fileno *no longer* returns a GEMDOS file handle, but instead an *internal* handle. The _chkufb function may be used to obtain the mapping from internal to GEMDOS handles.
- dcreat, dclose, dopen, dread, dwrite and dseek have had an underscore prefix added. Note also that the level of functionally at this level is significantly greater than before, as many of the GEMDOS anomalies are removed at this level.
- ^Z is no longer recognised as end of file marker in text files; this usage was not widespread and invariably caused confusion.
- isdata, isdptr, isstatic, isauto, isheap and ispptr were hang-overs from the MS-DOS implementation in version 3 and not strictly relevant to the 68000 environment and so have been removed.
- Almost all internal variables have been renamed or removed, any references to such variables should (in general) be removed.
- The values placed in _OSERR are now the positive GEMDOS error codes. This change was to increase conformity with other Lattice compilers.

- allmem and bldmem are no longer relevant to the new dynamic memory manager (cf. the old static memory manager); any calls to these functions are best removed.

- The exec family of functions are no longer available, they did not perform as described under the old documentation anyway (in fact they were fork synonyms). TOS is incapable of performing exactly the operations required to implement these functions as previously described.

- The -n flag on the compiler has the opposite sense to that under 3.04, i.e. -n *truncates* to eight characters rather than increasing the significance to 32 characters.

- The Line-A functions have *all* been implemented, and a correct header file made available together with many extensions. In particular, the showmouse macro from linea.h has been corrected to include the hide depth parameter; refer to the Line-A documentation for details.

- In conformance with ANSI, memcpy is not guaranteed to perform correctly when blocks overlap. Under version 3 this case was handled correctly.

- _MNEED is defunct due to the dynamic memory model used. It may be used to specify an initial heap size (e.g. if you have a program which will have only very small malloc requirements) although it no longer specifies the maximum heap size. In particular negative values are ignored.

- The GEM AES and GEM VDI header files have been split into two, although both may be included via #include <gemlib.h>.

- Many of the types in the AES and VDI prototypes have been changed to reflect more natural type usage. This should not, however, affect the code generated.

- The AES, VDI and Line-A libraries were re-written from scratch, hence any bugs in the old libraries are most unlikely to exist in the new libraries.

Other Differences

This section lists other differences from version 3; hopefully all important conversion differences are listed above. Here is the list of the improvements and changes in Version 5, relative to Version 3:

- The compiler now uses sequence points to ensure correct evaluation and side effect generation according to the standard.

- The compiler now includes a full ANSI pre-processor with string facilities, token facilities and appropriate scoping of substitution symbols. The defined() directive is also supported. In addition, __DATE__ and __TIME__ provide the date and time of compilation.

- The const and volatile keywords are supported.

- Function prototypes may now include an optional parameter name. Also functions taking a variable number of arguments may be indicated with ellipses '...'.

- String literals may now be concatenated to allow easier coding of long strings.

- The cast operation (void *) correctly coerces a type without any warning.

- Many diagnostic messages have been added to detect programs that do not conform to the ANSI standard.

- The compiler now recognises several new keywords:

signed	Overrides any default unsigned options.
near	Declares a data item to be addressed relative to the global base register. When used with a subroutine, it indicates a PC-relative subroutine call.
far	Declares an item that must be addressed with a full 32 bit address.
huge	Same as far.
__regargs	Defines a subroutine that is to be called with register parameters.
__stdargs	Defines a subroutine that is to be called with standard stack parameters.
__asm	Defines a subroutine that takes its parameters in a specific register
__saveds	Defines a subroutine that is to load up the global base pointer upon entry.
__interrupt	Defines a subroutine that may be called from interrupt code.

- To provide faster compilation of a large project, the symbol table may be saved out to disk so that it can be used in future compilations. If you run a large header file through the compiler and save it in this way, subsequent compilations using that header file will be much faster.

- The compiler provides an option to ignore redundant #include statements (i.e. several #include statements that refer to the same file).

- The search rules for #include files have been modified to conform to standard UNIX search conventions.

- It is now possible to disable particular error messages as well as to change the severity of most messages. Along with this, it is now possible to specify a maximum number of errors allowed in a compilation so that the compiler will abort.

- We have implemented an improved form of error recovery for many of the common mistakes to eliminate many of the situations that resulted in the cascade of errors.

- All of the compiler messages have been moved to a separate file to simplify adaptation of the compiler to non English language environments.

- In order to produce a compiler that fits well on a smaller system, we have elected to provide a big version of the compiler that includes some additional features. If you wish to take advantage of these features (at a cost of about 15K), you must use this big compiler instead of the standard one.

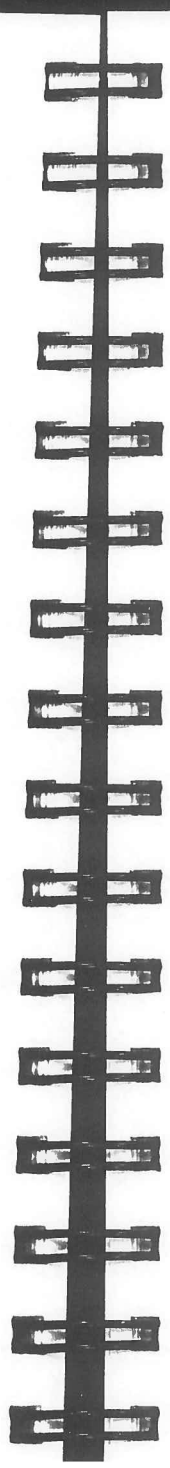
- The big compiler provides a full listing ability including macro expansion display, nest level counting, and include file listing. This listing may also include an optional cross reference of all variables, #define values and structure tags.

- The big version of the compiler may be used to generate prototype files of all functions encountered in a module. This eliminates the potentially tedious task of constructing the list of prototypes for all functions in a project.

- The compiler generates instructions optimised for each of the 680x0 family processors including support for the address modes found on the 68020 and 68030.

- The compiler provides an option to generate in-line floating point instructions which directly access the 68881 and 68882 maths co-processors. This code takes advantage of register tracking.

- You may now instruct the compiler to choose code sequences optimised for space or for time.



- In addition to the -r0/-r1 flags, you may freely mix the style of subroutine calls with the near and far keywords. Those declared near will be referenced with the pc-relative addressing while far will use the full 32-bit addressing.

- Data may be addressed much more freely with the near and far keywords. These control the type of addressing to be used for external data. Only those items declared near will be addressed as a 16-bit offset from the A4 register. All others will be accessed with a full 32-bit address.

- When the -cs option is used, string constants will be placed in the code section. This option is beneficial when using the -b0 option.

- Two styles of register parameters are supported. The -rr option causes the compiler to place, automatically, up to four parameters in registers for subroutine calls. The __asm keyword may be used in conjunction with a register specification list to cause the compiler to pass parameters in a given register.

- The compiler now tracks the condition codes affected by the generated code in an attempt to avoid generating unnecessary test instructions.

- Stack cleanup on subroutine calls is delayed as long as possible to allow coalescing and even elimination of the cleanup across multiple calls.

- The bitwise Boolean operations generate better code for dealing with constant values.

- Division by 2, 4, or 8 no longer generates a subroutine call. The compiler generates inline code to normalise and perform the calculation.

- Bit shift operations have been re-written completely. The compiler now generates optimal shift sequences for all constant values.

- The compiler attempts to place as many variables as possible in registers unless this feature is explicitly disabled.

- The code generator now takes full advantage of *all* 68000 address modes including auto increment and PC-relative indexed. Tracking of indexing operations allows the compiler to suppress unnecessary additions and substitute indexed address modes.

- Loading of specific constants has been optimised to generate the optimal code sequence and avoid MOVE.L # as much as possible.

- Several new built-in functions have been added:

abs	Return the absolute value of an integer.
__emit	Insert a hex word into the instruction stream.
..._fpc	Generate MC68881 transcendental operation.
geta4	Force loading of the global data register.
getreg	Obtain the contents of a specific 68000 register.
max	Return the larger of two integers.
memcmp	Compare memory blocks.
memcpy	Copy memory block.
memset	Initialise memory block.
min	Return the smaller of two integers.
putreg	Directly store into a specific 68000 register.
strcmp	Compare strings.
strcpy	Copy strings.
strlen	Obtain string length.

- Many small code optimisations suggested by users have been implemented. In particular, the compiler no longer will generate NOP instructions. Also, MOVEM instructions in the prologue/epilogue that reference a single register are converted to MOVE instructions.
- All GEMDOS/BIOS/XBIOS functions are available via inline TRAPs eliminating the overhead of 'stub' based methods.
- switch statements on values which are in the range of a short are converted to use the more efficient code.
- The entire run-time library has been re-written to take advantage of better algorithms. The most important routines were recoded in assembly language.

HiSoft C

Converting HiSoft C programs to Lattice C 5 is normally fairly simple, requiring few changes to the source code. The following options are recommended to ensure compatibility with the HiSoft C model:

-au Force all characters to unsigned; HiSoft C always considers plain characters to be unsigned. The use of this option will ensure that any dependencies on this behaviour will not cause unexpected effects.

-fd Force all floating point declarations to be of type double as HiSoft C only supports the double real format.

The following points should also be taken into account when converting code:

- If you are using the HiSoft C toolbox the main function should be renamed as lc_main, so that the special initialisation code runs before your program.
- The HiSoft C toolbox is supplied already compiled as a library for use with Lattice C 5 and can simply be linked in - see the **Installation Guide**.
- HiSoft C programs which do not use the GEM toolbox, but do use GEM, *must* use appl_init and appl_exit. Their use is optional under HiSoft C, whereas Lattice C 5 uses these calls to perform some initialisation itself. Failure to call these will result in mysterious crashes.
- The HiSoft C functions strgetfn and strsplfn were re-named from their traditional Lattice names. The functions strmfn and strsfn respectively should be used in their place. The easiest solution is to define this mapping on the command line to Lattice C (since HiSoft C would not allow the mapping within the program) viz:

```
-dstrgetfn=strmfn -dstrsplfn=strsfn
```

Where code cannot be made to run successfully under both systems, the IC and LATTICE pre-processor symbols can be used to distinguish between the translation environment; HiSoft C always defines IC, Lattice C 5 always defines LATTICE. Hence one can write:

```
#ifdef IC
  /* HiSoft C specific code */
#else
  /* Lattice C version of the code */
#endif
```

Appendix D
GST Support

LinkST, The GST format linker

Introduction

LinkST is a linker that links GST-format files. In general we recommend that you use the Lattice C Linker,CLink rather than LinkST since CLink is faster and has more facilities. LinkST does have one advantage over CLink though: it will let you link with other languages that produce GST format files.

For example, if you have some assembly language written with GenST, the assembler supplied as part of HiSoft DevpacST2, then you could use GenST to produce a GST-format file and then link it, together with your C code and the GST libraries, to produce an executable program. If the assembly language module you have is short or doesn't use the extensions to the Motorola standard provided by DevpacST, we recommend that you convert your assembly language to the Lattice assembler, asm, so that you *can* use CLink.

If you wish to link your Lattice C code with code written in another high level language that supports the GST format (producing a so-called *mixed language* program) or you wish to use a third party library provided in GST format, then you will need to use LinkST.

Producing mixed language programs is not for the faint hearted; you need to ensure that the required start up code is provided for both languages and that the memory models, including register conventions,are compatible. Fortunately Lattice C 5 provides a wealth of options, so that you can match most popular conventions. See the section on **LC, The Compiler**.

If you are 'only' linking with assembly language, things are much simpler since you can modify the code that is being called to handle whichever memory model you wish to use.

Note that Lattice C 5 does not produce the other commonly used format on the ST, DRI linkable code, because this linker format is not sufficiently rich to support some of code constructs that Lattice C 5 generates.

LinkST will only link GST-format files.

Compiling code in GST format

To produce code that can be linked with LinkST you will need to use the LC.TTP driver's -z option or, alternatively, convert a Lattice format file to GST format by using lc2gst as described at the end of this appendix.

If you are producing a program that is larger than about 32K using GST format then you cannot use the compiler's default small code model because GST linkers cannot generate ALVs, the clever method that CLink uses to allow you to use the small code model with large programs. Thus you will need to use the -r0 option when compiling GST-format libraries.

The GST libraries that we supply follow the same naming conventions as those in Lattice format only with the GST .bin extension. If you are using -r0, you'll need to use lcnb.bin, lcgnb.bin and lcmnb.bin as your libraries, which give you large code and large data. Thus to link with these you will also need to use the -b0 compiler flag.

There are a few other points regarding the use of GST format:

- The maximum amount of near data is restricted to 32K rather than 64K as with CLink.
- Lattice line number debugging information cannot be included in your executable program.
- There are some errors that CLink can spot that LinkST cannot; these are mainly mis-uses of memory models.

Invoking LinkST

As with most of the tools supplied with Lattice C 5 you can either run LinkST from the Desktop or from a shell and can either supply a complete command line or specify a link file which contains the required information.

The command line should be of the form:

```
<filename> <-options> [filename] [-options]
```

Options are denoted by a - sign then an alphabetic character, supported options being:

-B generate a true BSS section for any such named sections

-D debug - include all symbols in the binary file using DR standard 8 character format (for MonST2C or other debuggers)

-F force pass 2 of the linker, useful if you want to see all errors (as any pass 1 errors will, by default, stop the link before the second pass)

-L specify that all following filenames are library filenames

-M dump a map file showing the order of the sections and labels. The map filename will be the main filename with an extension of .MAP

-O specify object code filename, may be followed by white space before filename

-Q 'quiet' mode, which disables the pause after the link

-S dump a symbol table listing, The symbol table filename will be the main filename with an extension of .SYM

-T truncate to eight characters

-U force upper case

-W specify control file filename, defaults to .LNK extension

-X extended debug, using the HiSoft Extended Debug format for use with MonST2C.

Normally any file specified given are assumed to be input files, defaulting to the extension of .BIN, though if a .LNK extension is specified it will be taken to be a control file. After a -L option, filenames are all assumed to be library files.

The output file can be specified with the -o option on the command line, or using the OUTPUT directive in the control file. If there is more than one of these directives or options, the last one is used. If there is none given, then the first input filename specified in the command line or control file is used, with an extension of .PRG.

Example Command Lines

```
PART1 PART2 -d
```

Reads PART1.BIN and PART2.BIN as input files, and generates PART1.PRG as an output file complete with debugging information.

```
PART1 PART2 -o TEST.PRG
```

Reads PART1.BIN and PART2.BIN as input files, and generates TEST.PRG as an output file.

```
-o TEST.TOS START -l MYLIB -s
```

Reads START.BIN as an input file, selectively reads MYLIB.BIN as a library, and generates the output file TEST.TOS and the symbol listing file TEST.SYM.

LinkST Running

LinkST has two passes - during pass 1 it builds up a symbol table of all sections and modules, and during pass 2 it actually creates the output file. When it starts it prints a logon message, then reports on which files it is reading or scanning during both passes. This gives you some idea of what takes time to do, as well as exactly where errors have occurred.

If there is enough free memory at the end of pass 1, LinkST will use a cache to store the output file, which speeds up the process greatly. If it uses the cache it will write to the disk at the end of pass 2, and report the number of errors.

When the link finishes you will be prompted to press a key before quitting. This is to give you an opportunity to read any warning or error messages before returning to the Desktop. You can disable this pause by using the -q option, useful if you are using a CLI .

Error and warning messages are directed to the screen - if you want to pause output you can press Ctrl-S, while Ctrl-Q will resume. Pressing Ctrl-C will abort the linker immediately.

You can re-direct screen output to a disk file by starting the command line with

```
>FILENAME.TXT
```

or you can re-direct it to a printer by starting the command line with

```
>PRN:
```

to the parallel port, or

```
>AUX:
```

to the serial port.

If you do re-direct output in this way you should use the -q option as you won't be able to see the prompt at the end of the linking.

Control Files

The alternative way to run the linker is to have a control file for the programs which you are linking together.

If you require a lot of options which won't fit on the command line or you get bored of typing them, you can use a control file, which is a text file containing commands and filenames for the linker. The default extension is .LNK and the control filename is specified on the command line using the -w (for With) option. Each line can be one of the following:

```
INPUT <filename>
```

This specifies a filename to be read as an input file. The default extension is .BIN if none is given.

```
OUTPUT <filename>
```

This specifies the filename to be used for the output file. There is no default extension - you must specify it explicitly.

```
LIBRARY <filename>
```

This specifies a filename to be scanned as a library. The default extension is .BIN if none is given.

```
SECTION <sectionname>
```

This allows specific section ordering to be forced.

```
DEBUG
```

All symbol names included in the link are put in the output file so that debugging programs such as MonST2C can use them when the program is running.

```
XDEBUG
```

Similar to the debug option but uses HiSoft Extended Debug format for up to 22 character significance.

```
DATA size[K]
```

The BSS segment size is set accordingly. The size can be given either as a number of bytes or as a number of K-bytes (units of 1024). This option is particularly useful for the Prospero compilers which effectively use the BSS segment for their stack.

```
BSS <sectionname>
```

Specifies that the named section should lie in the GEMDOS BSS section area. This can save valuable disk space, but will generate errors if the section contains any non-zero data. This should not be used at the same time as the DATA statement.

```
TRUNCATE
```

Causes all symbols to be truncated to 8 characters. This is sometimes required to link assembly language with long labels to high-level language code with short labels.

```
UPPER
```

Forces all symbols to be automatically upper-cased. This is sometimes required when a compiler or assembler generates case-insensitive code.

Blank lines in the control file are ignored, and comments can be included by making the first character in the line a *, a ; or a !.

With the INPUT or OUTPUT directive, if the filename is specified as * it is substituted the first filename on the command line. This can be useful for having a generic control file for linking C programs for a particular memory model.

An example control file is:

```
* control file for linking large model gem programs
INPUT CNB
INPUT *
XDEBUG
LIBRARY LCGNB
LIBRARY LCNB
SECTION TEXT
SECTION DATA
BSS UDATA
```

Assuming this control file is called CPROG.LNK, the LinkST command line

```
TEST -w CPROG
```

will read as input files CNB.BIN and TEST.BIN, and scan the libraries LCGNB.BIN and LCNB.BIN. The object code, including extended debug information, will be written to TEST.PRG, as no output file was explicitly specified.

The two SECTION directives, above, ensure that the TEXT and DATA sections appear in the correct order in the output file. The BSS directive ensures that the UDATA section is treated as a true BSS section.

If you do not specify a drive name in the control file or on the command line, the default drive will be assumed. If you run LinkST from the Desktop, the default drive will always be the same as that containing the file on which you double-clicked; though if you run it from a CLI or from the editor this will not necessarily be so.

LinkST Warnings

Warnings are messages indicating that something might be wrong, but probably nothing too serious.

duplicate definition of value for symbol <x>

The symbol was defined twice. This can happen if you replace a subroutine in a module with one of your own, for example. The linker will use the first definition it comes across, and give this warning on the second.

module name is too long

Module names can only be 80 characters long.

comment is too long

Comment directives are only allowed to be 80 characters long (don't ask us why, we don't know!).

absolute sections overlap

Two absolute sections clash with each other.

SECTION <x> is neither COMMON nor SECTION

A section name was specified without defining its type.

LinkST General Errors

unresolved symbol <x> in file <x>

The symbol was referred to but not defined in the file. There may also be other files which refer to the symbol, but this error gives you a start in your search!

XREF value truncated

A value was too large to fit into the space allocated for it, for example a BSR to an external may be out of range.

bad control line <x>

An illegal line was found in a control file.

non-zero data in BSS section

A section wanted as a true BSS section contained non-zero data.

LinkST Input/Output (I/O) Errors

file <x> not found

Can't open output file <x>

Can't open map file <x>

Can't open symbol file <x>

Can't open input file <x>

i/o error on input file

disk write failed

filename <x> was too long

LinkST Binary File Errors

These are errors in the internal syntax of the input file, and should not occur. If they do it probably means the compiler, assembler or converter produced incorrect code.

missing SOURCE directive

Can occur if a file is not in GST format, for example a DRI file.

runtime relocation is only available for LONGs

attempt to redefine id of symbol <x>.

attempt to DEFINE <x> with <id> of zero

bad operator code 0x99 in XREF directive

bad truncation rule in XREF

wrongly placed SOURCE directive

bad directive <99>

<id> <99>not DEFINEd as a SECTION but used as one

attempted re-use of <id> <99>as SECTION id

attempted re-use of <x> as SECTION name

section is COMMON but being used as though it's not

SECTION is being misused as COMMON

unexpected end of input file

'Linker Bug' Messages

These can be produced as a result of internal checks by the linker. If you get one please send us copies of the files you are trying to link!

GSTlib, The GST format librarian

GSTlib is a librarian that is designed for maintaining GST format libraries. When specifying filenames to GSTlib you must explicitly include the file extension, normally .bin.

GSTlib has a number of different possible command lines as shown below:

Replace modules

```
gstlib r[vsq][a|b obmod] library [files...]
```

This will replace any current occurrences of the modules contained in the given files. If a module is not already present in the library then it will be added. If the library does not exist then a new library will be created that just contains these files. The following additional options may be used

v	(verbose). Echo when adding or replacing a module or creating a library.
a mod	(after). If adding a new module insert it after mod.
b mod	(before).If adding a new module insert it before mod.
s	(sort). Sort the library so that it can be scanned by a linker in a single pass, if possible. If this fails due to circular references then the new library will be saved in libname.tmp and the original library left as it was.
q	(quit). Wait for Return to be pressed before exiting GSTlib, for use with the Desktop.

Update modules

```
gstlib u[vsq][a|b obmod] library [files...]
```

This works in precisely the same way as the replace modules option except that the modules are only updated if the versions in the files are more up to date than the version in the library. Exactly the same options may be used as with the r option.

Load modules

```
gstlib l[vq] prog.lib [files...]
```

This option can be used to produce a library that contains just the modules that would be included when linking a program. The v and q modifiers have their usual meaning. For example:

```
gstlib l prog.bin c.bin yourprog.bin lc.bin
```

could be used to create a library containing the modules required by yourprog. You could then run LinkST without the need to scan the library. Once you have created a library using this option it is unwise to sort it subsequently.

Delete modules

```
gstlib d[vsq] library [modules....]
```

Deletes the modules with the given names. As usual v will cause the librarian to inform you of its progress, q will pause before exiting and s will attempt to sort the library after performing the deletions.

Move modules

```
gstlib m[vsq][a|b obmod] library [modules...]
```

Moves the given modules to thr end of the library unless either a or b are specified, in which case:

a mod	moves the modules immediately after obmod and
b omod	moves the modules immediately before obmod.

The other modifiers have their usual meanings.

Tabulate modules

```
gstlib t[vvvvvq] library [modules...]
```

This form of command line displays information about the given modules in the library. If no module list is included, the information is given about all modules. The different levels of information are as follows:

v	module names only
vv	as per v and the size and date
vvv	as per vv and the list of exported (xdef) symbols

vvvv as per vvv and the list of imported (xref) symbols

vvvvv as per vvvv and a cross reference of the symbols

As ever, q may be included to pause before GSTlib terminates.

Extract modules

```
gstlib x[vkq] library [modules....]
```

This extracts the given modules from the library. If no modules are specified then all the modules are extracted. Note that module names may have .o, .c or .bin appended to them depending on the tool that produced them. The additional modifiers are:

k keep date stamp from library on the extracted files

v inform the user of progress

q pause before exiting

If a module that is being extracted does not have a name it will be placed in a file called dummy###.BIN where ### is a unique decimal number.

Librarian command files

If the command line to GSTlib contains an @ sign, the name following this is taken as a command file name and the command read from this. Additionally such files may contain lines starting with #; these are treated as comments. Long lines may be split by using \ as the last character of the line.

Example command lines

```
gstlib tv lc.bin
```

List all the modules in the library lc.bin; there are quite a few!

```
gstlib xk lc.bin printf.o
```

Extract the module printf.o into a file called printf.o retaining the date stamp that it had in the library.

```
gstlib d lc.bin printf.bin
```

Remove the module printf.bin from lc.bin.

The source code to GSTlib is supplied; see the **Installation Guide** for details.

lc2gst, The Object File Convertor

lc2gst is a tool for converting Lattice C object files to the GST format. It has three different forms of command line as follows:

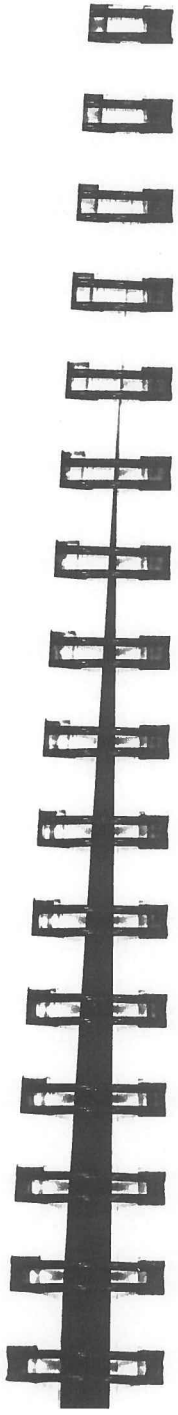
```
lc2gst file.o
lc2gst -o dir\ file.o
lc2gst -o john file.o
```

The first form simply converts file.o to file.bin. Note that the .o *is* required.

The second form places the converted file in the directory dir; note that the terminating \ must be present. This option can be used to keep your GST format files in one directory.

The final form causes file.o to be converted to john.bin. Note that the .bin extension is always used, regardless of any explicit extension after the -o.

lc2gst can convert more than one file at once: simply include the other files to convert at the end of the command line. It does not, however, expand wildcards itself, so that you will need a shell that supports this, such as Craft, to provide this facility.



Appendix E
Quick Options Reference

LC	LC1	LC2	
-b	-b		Base relative data
-b0	-b0		Non-base relative data
-b1	-b1		Base relative data
-B			Always use 'big' compiler
-c+	-c+		Suppress structure messages
-ca	-ca		ANSI compatibility
-cc	-cc		Allow nested comments
-cd	-cd		Allow $ in identifiers
-ce	-ce		Suppress error line printing
-cf	-cf		Require function prototypes
-cg	-cg		Process ANSI trigraphs (not implemented)
-ci	-ci		Suppress multiple includes of same file
-ck	-ck		Allow new keywords
-cl	-cl		Forces long alignment of all external data
-cm	-cm		Allow multiple character constants
-co	-co		Enable old style preprocessor
-cq	-cq		Strengthen aggregate equivalence type checking
-cr	-cr		Allow register keywords
-cs	-cs		Create only one copy of identical strings
-ct	-ct		Enable warnings for tags used without definition
-cu	-cu		Force all char declarations as unsigned char
-cw	-cw		Shut off warning for return without a return value
-cx	-cx		Treat all global declarations as externals
-C			Continue on error
-d	-d		Enable debugging
-d0	-d0		Disable debugging

-d1	-d1		Enable debugging - dump line table
-d2	-d2		Generate symbol information
-d3	-d3		Generate symbol information, dump at every line
-d4	-d4		Generate full symbol information
-d5	-d5		Generate full symbol information, dump at every line
-dx=y	-dx=y		Define preprocessor symbol
-e	-e		Recognise extended character set
-e0	-e0		Japanese character set
-e1	-e1		Chinese character set
-e2	-e2		Korean character set
-Ex=y			Define environment variable x with value y
-f	-f		Use standard Lattice libraries
-f8	-f8		Generate code for Motorola 68881
-fa	-fa		Auto-detecting I/O based 68881
-fi	-fi		I/O based 68881 maths
-fl	-fl		Use standard Lattice libraries
-fd	-fd		Treat all declarations as double precision
-fm	-fm		Use float as single precision and double as double precision
-fs	-fs		Treat all declarations as single precision
-gd	-gd		Cross reference defined symbols
-gc	-gc		Cross reference compiler provided files
-ge	-ge		List excluded lines
-gh	-gh		List header files
-gi	-gi		List included files
-gm	-gm		List macro expansions
-gn	-gn		Print narrow lines
-gs	-gs		List source
-gx	-gx		Produce cross reference listing
-Hxxx	-hxxx		Read in header file xxx
-ix	-ix		Specify include directory
-j<n>	-j<n>		Disable message n
-j<n>e	-j<n>e		Make message n an error instead of a warning

-j<n>i	-j<n>i		Disable message n
-j<n>w	-j<n>w		Enable message n as a warning
-L+			Specify additional linker objects
-La			XADDSYM option
-Lb			BATCH option
-Lf			MAP option
-Lg			GEM library
-Lh			Hunk map option
-Ll			Library map option
-Lm			Lattice maths library
-Ln			NODEBUG option
-Lq			QUIET option
-Ls			Symbol map option
-Lv			Verbose option
-Lx			XREF map option
-l	-l		Align objects on longword boundaries
-M			Only compile modified source files
-m		-m	Generate code for Motorola 68000
-m0		-m0	Generate code for Motorola 68000
-m1		-m1	Generate code for Motorola 68010
-m2		-m2	Generate code for Motorola 68020
-m3		-m3	Generate code for Motorola 68030
-ma		-ma	Generate code for all Motorola processors
-mc		-mc	Disable cleanup overhead reduction enhancement
-mr		-mr	Disable automatic registerisation
-ms		-ms	Generate code optimised for space
-mt		-mt	Generate code optimised for time
-n	-n		Retain only 8 characters for identifiers
-ox		-ox	Place object file in location x
-p	-p		Preprocess only
-pe	-pe		Generate prototypes only for externs
-ph	-ph		Generated precompiled header file

-pp	-pp		Generate prototypes with __PROTO for portability
-pr	-pr		Generate prototype file
-ps	-ps		Generate prototypes only for static functions
-qx	-ox		Place quad file in location x
-q<n>e	-q<n>e		Quit compilation after n errors/warnings
-q<n>w	-q<n>w		Quit compilation after n warnings
-q			Same as -q1e1w
-q-	-q-		Never quit on any errors or warnings
-r	-r		Default subroutine calls to near (PC-relative)
-r0	-r0		Default subroutine calls to far (Absolute)
-r1	-r1		Default subroutine calls to near (PC-Relative)
-rr	-rr		Default subroutine calls/entries to register conventions
-rs	-rs		Default subroutine calls/entries to stack conventions
-rb	-rb		Generate code for both register and stack convention entries
-Rx			Place compiled objects into library x
-s		-s	Specify default segment names
-sb=x		-sb=x	Specify name for BSS segment
-sc=x		-sc=x	Specify name for code segment
-sd=x		-sd=x	Specify name for data segment
-ta			Force linking of a desk accessory startup stub
-td			Force linking of auto detecting startup stub
-tr			Force linking of resident startup stub
-t=x			Use file x as the startup code
-u	-u		Undefine all preprocessor symbols
-ux	-ux		Undefine preprocessor symbol x
-v		-v	Disable stack checking code
-w	-w		Default to short integers
-x	-x		Treat all global declarations as externals
-y		-y	Load up A4 with base address at start of functions
-z			Generate GST-linkable code

Appendix F
The Lattice C Start-Up

Introduction

The Lattice C compiler is supplied with a wide range of differing startup stubs which perform various levels of initialisation and have differing impacts on the programs which may be written.

The stubs

4 different stubs are provided for the following purposes:

Standard - suffix none

This family of stubs provide the normal entry and exit code required by a GEM or TOS program designed to be run either from the Desktop or from another program via GEMDOS Pexec. They perform full command line parsing via either the Atari extended command line method, or if this is not available via the command line embedded in the programs basepage. The memory available to the program is assessed and various internal variables are initialised to support the dynamic memory allocator (note that this allocator is careful to ensure that bugs in TOS pre-1.4 are not aggravated). The environment variables available are also parsed into the standard UNIX environ format.

The final call, from the startup code, is to the pre-main routine _main which initialises the standard I/O library prior to calling the main routine. If your program uses no standard I/O you may wish to declare your main function as _main to ensure that no superfluous library routines are drawn in.

Desk Accessory - suffix 'acc'

The desk accessory family of stubs perform the minimal initialisation required by desk accessories. The only 'normal' operations performed are the initialisation for the clock function and the reading of the OS information. The stack space for a desk accessory is fully contained within the startup code, so if a different stack size is required this must be done by modifying the stack size in the startup and reassembling it, the relevant line is:

```
base        ds.b        256
```

which is normally near to the last line in `c.s`.

These stubs perform no command line parsing, no environment setup, no standard file opening and *no* dynamic memory allocator initialisation. This last point is so that all DAs are forced to conform to the requirement that a DA may not legally use GEMDOS Malloc.

In order to circumvent the malloc problem you may add static arrays to the library free memory list using the _addheap function. The function has the prototype:

```
void _addheap(void * base,size_t length);
```

Calling this function with the base of a static array and and a length count adds those bytes to the library heap. For example:

```
int main(void)
{
        static long heap[100];

        _addheap(heap,sizeof(heap));
        ...
}
```

Note that the length of the heap *must* be a longword multiple. Also be aware that if you never call this function and your program calls *any* function which must malloc() memory (e.g. fopen()) then all such calls *will* fail.

Because desk accessories are always GEM programs they must always be linked with the GEM library, additionally the DA startup is specified using the -ta option.

Auto-detecting - suffix 'aut'

This family of stubs allow a program to detect how it has been run, from the auto-folder, as a desk accessory, as a GEM program or as a TOS program. An external variable, int _XMODE, is made available so that the program may perform the requisite initialisation:

0	Standard GEM program
1	Standard TOS program
2	GEM DA
3	Auto folder TOS program

For the GEM/TOS modes (0, 1 and 3) the startup code performs otherwise as described for a normal GEMDOS program. For the DA mode, the stack issue recurs, in this instance the startup code re-uses the space taken by the startup code as the stack for the desk accessory. This gives approximately 750 bytes of stack space, which may be increased by inserting NOPs, immediately before the lsada2 label. Other than this oddity, the DA startup sequence is identical to that described previously.

Resident - suffix 'res'

The resident stubs provide a facility for generating programs which are both residentable and re-entrant. This is used, for example, by LC2 so that it may be multiply re-executed from within the integrated environment. The resident stubs place no special restrictions on the programmer other than that all accesses to the data must be referenced through A4, i.e. the normal -b1 model. Note that these means there must be *no* far data whatsoever.

In the resident mode the startup code initially detects whether or not the program is being executed in a resident manner, or whether it has been run normally, as if by Pexec(0, ...). A copy of the near data section is then made so that any relocation required may be performed on it, prior to initialising A4. Note that in this mode the default locations of any global data variables exported for a symbolic debugger point to the original constant copy of the data and not the copy which is in use.

To use a residentable program from within your own application, it must initially be loaded using Pexec(3, ...) and then shrunk viz:

```
#include <osbind.h>
#include <basepage.h>

bp=(BASEPAGE *)Pexec(3,prog,"","");
Mshrink(bp,bp->p_tlen+bp->p_dlen+0x100);
Mfree(bp->p_env);    /* kill the environment */
```

To execute this program, another basepage must be allocated for the active copy, and then executed:

```
BASEPAGE *ap=(BASEPAGE *)Pexec(5,NULL,command,env);

ap->p_tbase=bp->p_tbase;
ap->p_tlen=bp->p_tlen;
ap->p_dbase=bp->p_dbase;
ap->p_dlen=bp->p_dlen;
ap->p_bbase=ap+1;
ap->p_blen=(long)ap->p_hitpa-(long)ap->p_lowtpa-sizeof(*ap);
err=Pexec(4,NULL,(char *)ap,NULL);
Mfree(ap->p_env);
Mfree(ap);
```

where command and env give the command line and environment respectively that are to be passed to the child process.

If a residentable program is to be executed normally a standard Pexec(0, ...) suffices. Please note that a residentable program whilst conforming to the normal GEMDOS program load format contains many extensions which are generated by the linker in order to support the resident concept. At this time it is *not* our intention to provide detailed information on this since such information is only useful to the startup stubs (provided) and the linker, CLink.

User supplied stubs

A user specified stub may be passed, by the user, to the compiler driver. Typically this is used when the startup code has been modified for a special purpose, or the stack size of a D.A. has been increased, for example.

The name of the stub is communicated to the driver via the -t= option, in conjunction with the -L options. Hence to compile and link with the stub mystub.o:

```
lc -L -t=mystub.o myprog.c
```

Naming conventions

The naming conventions for the startup stubs follows exactly the same pattern as that for libraries. The first character of a startup stub is always c, followed by:

Letter	Type of library
s	Default short integer (-w)
r	Register parameter passing (-rr)
nb	Non-base relative (-b0)

The final three character suffix, as described above, are as follows:

Suffix	Type of startup stub	Option
acc	Desk accessory	-ta
aut	Auto program type detecting	-td
res	Resident program	-tr
User	User specified stub	-t=

Hence a file called `csrnbacc.o` would be a short integer, register passing non-base relative desk accessory startup stub.

Re-assembling c.s

The startup code is provided so that it may be modified as the user requires. To re-assemble it the command:

```
asm [options] c.s
```

should be used, either from a shell, the desktop or as a Tool/Run Other command from the environment.

The options part dictates what sort of stub is to be produced, all types of stub are controlled using various -d parameters. These are:

AUTO	Auto-detecting (-td)
DA	Desk accessory (-ta)
GST	GST compatible (-z)
NOBASER	Non-base relative mode (-b0)
REGARGS	Register passing mode (-rr)
RESIDENT	Residentable program (-tr)
SHORTINT	Default short integer mode (-w)

Hence to build the stub described earlier, `csrnbacc.o`, the command used is:

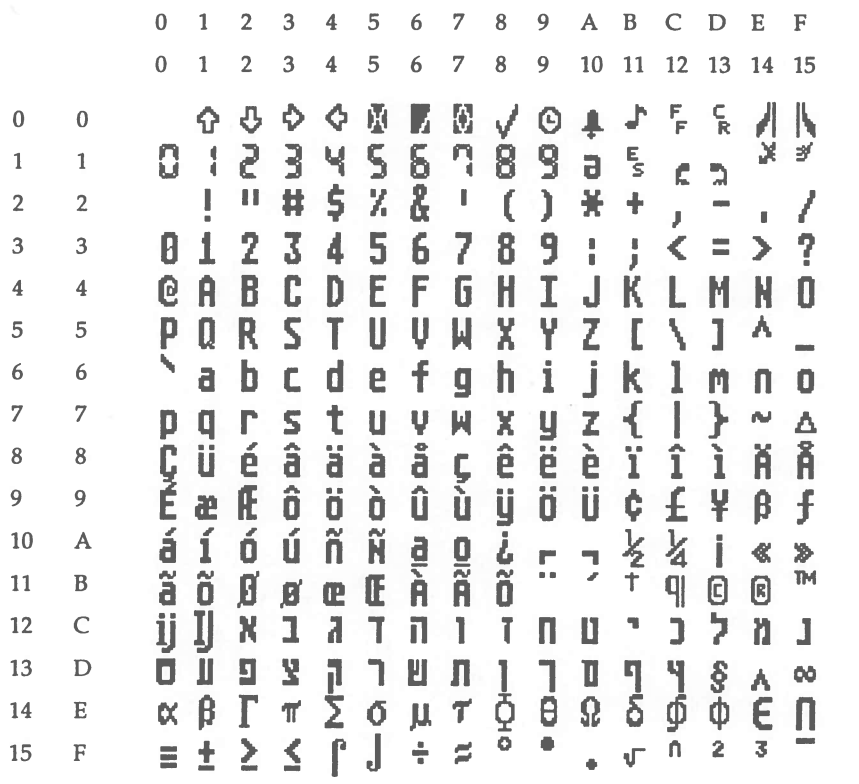
```
asm -dDA -dREGARGS -dNOBASER -dSHORTINT c.s
```

Note that this will in fact build the linkable file into `c.o` rather than `csrnbacc.o`.

Also note that specifying GST does not build a GST compatible linkable file, the `.o` produced by asm must be passed through lc2gst first.

Appendix G
ST ASCII Table

Here is the 8-bit ASCII representation of the ST's character set:



		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0																
1	1																
2	2																
3	3																
4	4																
5	5																
6	6																
7	7																
8	8																
9	9																
10	A																
11	B																
12	C																
13	D																
14	E																
15	F																

The most significant four bits of the ASCII representation are shown down the left side whereas the least four significant bits are across the top so that, for example:

4C (4*16+12=76 decimal) represents **L**

7B (7*16+11 = 123 decimal) represents **{**

Appendix H
VT52 Screen Codes

When writing to the screen via GEMDOS or the BIOS calls, the screen driver emulates VT52 protocols. The control codes are sent via *escape* sequences, which means an escape character is sent (27 decimal, or $1B) followed by one or more other characters.

ESC A	Cursor up; no effect if at the top line
ESC B	Cursor down; no effect if at the bottom line
ESC C	Cursor right; no effect if on the right hand side
ESC D	Cursor left; no effect if on left hand side
ESC E	Clear screen and home cursor
ESC H	Home cursor
ESC I	Move cursor up one line; if at top scrolls the screen down a line
ESC J	Erase to end of screen, from the cursor position onwards
ESK K	Clear to end of line
ESC L	Insert a line by moving all following lines down. Cursor is positioned at start of the new line
ESC M	Delete a line by moving all following lines up
ESC Y	Position cursor; should be followed by two characters, the first being the Y position, the second the X. Row and column numbering starts at (32, 32) which is the top left
ESC b	Foreground colour; should be followed by a character to determine the colour, of which the four lowest bits are used
ESC c	Background colour; similar to above
ESC d	Erase from beginning of display to the cursor position
ESC e	Enable cursor
ESC f	Disable cursor
ESC j	Save the current cursor position
ESC k	Restore a cursor position saved using ESC j; note that this is not supported on the original 1.0 ROMs
ESC l	Erase a line and put cursor at start of line
ESC o	Erase from start of line to cursor position
ESC p	Inverse video on
ESC q	Inverse video off
ESC v	Wrap around at end of line on
ESC w	Wrap around at end of line off

For instance these codes may be used in a normal printf statement to print information one after another, on the same line, e.g.

```
printf("\033IProcessing %s\033K\n",file);
```

Appendix I
Bibliography

C Programming

Advanced C **Schildt, Herbert (1986)**

ISBN 0-07-881208-9, McGraw-Hill, Berkeley, CA 94710, USA.

Advanced C Techniques & Applications
Sobelman, Gerald E. and David E. Krekelberg (1985)

ISBN 0-88022-162-3, QUE Corporation, Indianapolis, IN 46250, USA.

Advanced C: Food For The Educated Palate **Gehani, Narain (1985)**

ISBN 0-88175-078-6, Computer Science Press, Rockville, MD 20850, USA.

ANSI C: A Lexical Guide **Mark Williams Company (1988)**

ISBN 0-13-037814-3, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

ANSI X3.159-1989 - Programming Language - C **ANSI (1990)**

American National Standards Institute, 1430 Broadway, New York 10018, USA.

C For Beginners **Sinclair, Ian (1986)**

ISBN 0-86161-206-X, Melbourne House Publishers Ltd., 60 High Street, Kingston-Upon-Thames, Surrey KT1 4DB, U.K.

C Programming Guide **Purdum, Jack (1983)**

ISBN 0-88022-022-8, QUE Corporation, Indianapolis, IN 46250, USA.

C Self-Study Guide **Purdum, Jack (1985)**

ISBN 0-88022-149-6, QUE Corporation, Indianapolis, IN 46250, USA.

C Wizard's Programming Reference **Schwaderer, W. David (1985)**

ISBN 0-471-82641-3, Wiley Press, New York, NY 10158, USA.

C: A Reference Manual, 2nd Edition
Harbison, Samuel P. and Guy L. Steele Jr (1984)

ISBN 0-13-109802-0, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Common C Functions — **Brand, Kim Jon (1985)**
ISBN 0-88022-069-4, QUE Corporation, Indianapolis, IN 46250, USA.

Debugging C — **Ward, Robert (1986)**
ISBN 0-88022-261-1, QUE Corporation, Indianapolis, IN 46250, USA.

Efficient C — **Plum, Thomas and Jim Brodie (1985)**
ISBN 0-911537-05-8, Plum Hall, Cardiff, NJ 08232, USA.

Going from BASIC to C — **Traister, Robert J. (1985)**
ISBN 0-13-357799-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Introducing C — **Allen, Boris (1986)**
ISBN 0-00383-105-1,William Collins Sons & Co. Ltd., 8 Grafton Street, London W1X 3LA, U.K.

Learning to Program in C — **Plum, Thomas (1983)**
ISBN 0-911537-00-7, Plum Hall, Cardiff, NJ 08232, USA.

Microsoft C Run-Time Library — **Jamsa, Kris (1985)**
ISBN 1-55615-227-2, Microsoft Press, Bellevue, WA 98009, USA.

Programming In C For The Microcomputer User — **Traister, Robert J. (1984)**
ISBN 0-13-729641-X, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Software Engineering in C — **Darnell, Peter A. and Philip E. Margolis (1988)**
ISBN 0-387-96574-2, Springer-Verlag, 175 Fifth Avenue, New York, NY 10010, USA.

Standard C — **Plauger, P. L. and J. Brodie (1989)**
ISBN 1-55615-158-6, Microsoft Press, 16011 NE 36th Way, Box 97017, Redmond, Washington, 98073-9717, USA..

The C Compendium — **Lawrence, David and Mark England (1985)**
ISBN 0-946498-86-6, Sunshine Books, 12-13 Little Newport Street, London WC2H 7PP, U.K.

The C Library — **Jamsa, Kris (1985)**
ISBN 0-07-881110-4, McGraw-Hill, Berkeley, CA 94710, USA.

The C Programmer's Handbook — **Bolsky, M.I. (1985)**
ISBN 0-13-110073-4, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The C Programmer's Handbook — **Hogan, Thom (1984)**
ISBN 0-89303-365-0, Brady Communications Company, Inc., Bowie, MD 20715, USA.

The C Programming Language, 2nd Edition — **Kernighan, Brian W. and Dennis M. Ritchie (1988)**
ISBN 0-13-110370-9, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The C Toolbox — **Hunt, William James (1985)**
ISBN 0-201-11111-X, Addison-Wesley Publishing Company, Reading, MA, USA.

UNIX System V Programmers Reference Manual — **AT&T (1987)**
ISBN 0-13-940479-1, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Variations In C — **Schustack, Steve (1985)**
ISBN 0-914845-48-9, Microsoft Press, Bellevue, WA 98009, USA.

68000

16-Bit Microprocessors — **Whitworth, Ian R. (1985)**
ISBN 0-00-383113-2, William Collins Sons & Co. Ltd., 8 Grafton Street, London W1X 3LA, U.K.

68000 Assembly Language Programming 2nd Edition — **Kane, G., D.Hawkins and L.Leventhal (1987)**
ISBN 0-07-881232-1, Osborne/McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710, USA.

68000 Machine Code Programming — **Barrow, David (1985)**
ISBN 0-00-383163-9, William Collins Sons & Co.Ltd., 8 Grafton Street, London W1X 3LA, U.K.

68000, 68010, 68020 PrimerKelly-Bootle, Stan and Bob Fowler (1985)
ISBN 067-22405-4, Howard W.Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

M68000 Family Programmer's Reference Manual
Motorola Inc. (1989)
Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

Mastering The 68000 Microprocessor **Robinson, Phillip R. (1985)**
ISBN 0-8306-1886-4, Tab Books Inc., Blue Ridge Summit, PA 17214, USA.

Microprocessor Systems: A 16-Bit ApproachEccles, William J. (1985)
ISBN 0-201-11985-4, Addison-Wesley Publishing Company, Reading, MA, USA.

Programming the 68000 **Williams, Steve (1985)**
ISBN 0-89588-133-0, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

The MC68000 User's Manual 7th Edition **Motorola Inc. (1989)**
ISBN 0-13-567074-8, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The MC68020 User's Manual 2nd Edition **Motorola Inc. (1985)**
ISBN 0-13-566878-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

The MC68030 User's Manual **Motorola Inc. (1987)**
Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

The MC68881/MC68882 User's Manual **Motorola Inc. (1987)**
ISBN 0-13-566936-7, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Algorithms & Data Structures

Algorithms in C **Sedgewick, Robert (1990)**
ISBN 0-201-51425-7, Addison-Wesley Publishing Company, Reading, MA, USA.

C Chest and Other C Treasures **Holub, Allen I. (1987)**
ISBN 0-934375-40-2, M & T Books, 501 Galveston Drive, Redwood City, CA 94063, USA.

Compilers: Principles, Techniques and Tools
Aho, Alfred V, Ravi Sethi and Jeffrey D. Ullman (1986)
ISBN 0-201-10194-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Data Handling Utilities In C
Radcliffe, Robert A. and Thomas J. Raab (1986)
ISBN 0-89588-304-X, Sybex, Berkeley, CA 94710, USA.

Data Structures and Algorithms
Aho, Alfred V, John E. Hopcroft and Jeffrey D. Ullman (1983)
ISBN 0-201-00023-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Fundamental Algorithms **Knuth, Donald E. (1973)**
ISBN 0-201-03809-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Seminumerical Algorithms **Knuth, Donald E. (1981)**
ISBN 0-201-03822-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Sorting and Searching **Knuth, Donald E. (1973)**
ISBN 0-201-03803-X, Addison-Wesley Publishing Company, Reading, MA, USA.

ST Specific

A Hitchhikers Guide to the BIOS **Atari Corp (1986)**
Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Atari GEMDOS Reference Manual **Atari Corp (1986)**
Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Atari ST Internals 3rd Edition
Brückmann, Rolf, Lothar Englisch and Klaus Gerits (1988)
ISBN 0-916439-46-1, Data Becker GmbH, Merowingerstraße 30, 4000 Düsseldorf, West Germany.

COMPUTE!'s ST Applications Guide: Programming in C
Field, Simon, Kathleen Mandis and Dave Myers (1986)
ISBN 0-87455-078-5, COMPUTE! Publications, Inc., P.O. Box 5406 Greensboro, NC 27403, USA.

COMPUTE!'s Technical Reference Guide, Atari ST - Volume I: VDI **Sheldon Leeman (1987)**
ISBN 0-87455-093-9, COMPUTE! Publications, Inc., P.O. Box 5406 Greensboro, NC 27403, USA.

Concise Atari ST 68000 Programmer's Reference **Katherine D. Peel (1986)**
ISBN 1-85181-017-X, Glentop Publishers Ltd., Standfast House, Bath Place, High Street Barnet, Herts EN5 5XE, U.K.

GEM Programmer's Guide, Volume 1: VDI **Digital Research (1987)**
Digital Research Inc., 60 Garden Court, P.O. Box DRI, Monterey, CA 93942, USA.

GEM Programmer's Guide, Volume 2: AES **Digital Research (1987)**
Digital Research Inc., 60 Garden Court, P.O. Box DRI, Monterey, CA 93942, USA.

GEMDOS Extended Argument (ARGV) SpecificationAtari Corp (1989)
Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Professional GEM **Oren, Tim (1985)**
ANTIC Publishing

Programmers Guide to GEM Balma, Phillip and William Fitler (1986)
ISBN 0-89588-297-3, SYBEX Inc., 2344 Sixth Street, Berkeley, CA 94710, USA.

Rainbow TOS Release Notes **Atari Corp (1989)**
Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

S.A.L.A.D. - Still Another Line A Document **Atari Corp (1987)**
Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

STE TOS Release Notes **Atari Corp (1989)**
Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

The Pexec Cookbook **Atari Corp (1989)**
Atari Corp, 1196 Borregas Avenue, Sunnyvale, CA 94086, USA.

Appendix J Technical Support

So that we can maintain the quality of our technical support service we are detailing how to take best advantage of it. These guidelines will make it easier for us to help you, fix bugs as they get reported and save other users from having the same problem. Technical support is available in five ways:

Phone our technical support hour is normally between 3pm and 4pm, though non-European customers' calls will be accepted at other times.

Post if sending a disk, *please* put your name & address on it.

BIX™ our username is (not surprisingly) *hisoft*. Would UK customers please use CIX or more old fashioned methods; it's cheaper for everyone.

CIX™ our username is (still not surprisingly) *hisoft*.

GEnie™ our username is ... OK, you've guessed it!

For bug reports, please always quote the program, computer, version number of the program (the one from the file READ.ME) and the serial number found on your master disk. If your problem is with the compiler please ensure that you tell us *all* compiler options which you used so that we can attempt to duplicate the problem.

If you think you have found a bug, try and create a small program that reproduces the problem. It is always easier for us to answer your questions if you send us a letter and, if the problem is with a particular source file, please enclose a copy *on* disk (which we *will* return).

Remember, technical support is *only* available to registered users i.e. those people that have filled out the registration card enclosed with Lattice C 5 and mailed it to HiSoft.

Upgrades

As with all our products, Lattice C 5 is undergoing continual development and, periodically, new versions become available. We normally make a small charge for upgrades, though if extensive additional documentation is supplied the charge may be higher. All users who return their registration cards will be notified of *major* upgrades.

Suggestions

We welcome any comments or suggestions about our programs and, to ensure we remember them, they should be made in writing.

Common Problems

The following is a list of common mistakes made when starting programming with Lattice C. Please check these carefully against your problem before ringin for technical support.

? Why does the linker complain that symbols beginning with __CX or __AES are undefined and wants me to enter a "DEFINE value"?

• These are symbols in the floating point maths library and/or GEM libraries (respectively), if you have used any floating point maths or GEM calls you must link with the relevant library. This involves selecting Link with floating point or Link with GEM from the options menu if using the integrated compiler, or using the -Lm or -Lg options from the command line.

? My program was nearly finished, but suddenly the compiler has started reporting "Intermediate file error". How can I stop this?

• This almost certainly means that the compiler has run out of space on the device on which it is creating the quad (intermediate) file. Either delete some files from the appropriate disk, or if it is a RAM disk, and you have enough memory left, increase its size.

? Whenever I try to read/write screen images, they always get corrupted. Why?

• The libraries support two modes of file operation, text (the default) and binary. In text mode all CR/LF pairs are considered special and may be removed/added by the library. If using a binary file (such as a screen image) make sure you have selected the binary mode of operation, see the page on fopen() etc. for more details on this.

? My program mysteriously crashes, and I think this is the compiler corrupting the stack. Is this a bug?

• What has probably happened is that you have either allocated too much stack space with stack checks disabled (these are *on* by default), or that you are writing over the end of an array which you have allocated on the stack. This problem can be hard to track down, so always make sure you test your program with stack checks on.

Index

■ B

■ C

D

■ E

F

G

H

I

J

K

L

■ M

■ N

■ O

S