Building interp

August 27, 2011

Before you can successfully build customized versions of interp, you must have a clear understanding of the configuration parameters (symbols) used during the build process. These parameters fall into one of three classes:

1. Parameters specifically for embedded builds,
2. Parameters that control memory usage for both embedded and Linux builds, and
3. Parameters that create feature-limited or special-purpose versions of interp (sin and I2.)

1. CONTROLLING INTERP'S FEATURES

Interp can be customized for non-Linux execution environments that include embedded RTOS and no OS at all. For these types of environments it is important to be able to restrict or eliminate the command line interface (CLI) and remove functions that make Linux system calls because their presence in the code will prevent compiling and linking.

1.1 "NOFLAGS" Parameter

If you are using specialized hardware (in-circuit or JTAG emulator) to load and execute interp, then the manufacturer's environment may allow you to pass parameters to interp. The "NOFLAGS" parameter removes support for all the command line options that start with a ""-" because you can't use them anyway, but still allows you to pass parameters to your interp program.

1.2 "NOCLI" Parameter

If you are running interp without any type of OS support (directly from reset) then you want the "NOCLI" flag which removes all CLI support.

1.3 "SLMODE" Parameter

At the same time that you are using the "NOFLAGS" or "NOCLI" parameters, you may also want to specify "SLMODE". When this symbol is defined, slib.c is compiled and used in place of two standard c library header files: ctype.h and string.h. You would only need to do this if your embedded environment doesn't provide their own version of these libraries. The functions that are needed by interp are isprint(), isgraph(), strlen(), strcpy(), memmove(), and memcpy(). The affected files are: idefine.c, imain.c, interp.c, interpio.c, and isin.c.

1.4 Porting Reminder

Don't forget that using the "NOFLAGS" or "NOCLI" parameters implies that you have written replacement code for specific functions in ilowlevelio.c (that do not make Linux systems calls.) Please refer to the porting monographs in DOC/PORTING for details.

1.5 "make" command line examples

make nocli	selects the "NOCLI" parameter, same as: make all NOCLI="-DNOCLI"
make slnocli	selects both "NOCLI" and "SLMODE", same as: make all NOCLI="-DNOCLI -DSLMODE"
make noflags	selects the "NOFLAGS" parameter, same as: make all NOFLAGS="-DNOFLAGS"
make slnoflags	selects both "NOFLAGS" and "SLMODE", same as: make all NOFLAGS="-DNOFLAGS -DSLMODE"

Also refer to the Makefile and the README file for additional information.

2. CONTROLLING INTERP'S MEMORY USAGE

Embedded builds for target systems with small RAM areas are obvious candidates for a custom interp build. Linux builds can also be customized with these parameters.

In the release Makefile, the definitions of the symbols are commented out with a "#" (in column 1.) If you wish to change one of these default settings, duplicate the line, uncomment the copy by deleting the "#", change the value you need, remove everything else from the line, save your change, and build interp.

After making your changes, use any of the command line examples show above or in the README file to build interp.

2.1 Macro Processor Configuration Parameters (MACRO_OPTIONS)

If an interp application runs out of macro definitions, you need to increase the number of macro definitions that can be made. The macro table is allocated at compile time (when interp is built) and cannot be expanded without rebuilding.

SYMBOL NAME	DEFAULT
BUFSIZE	1024
MAXPTR	128
MAXDEF	71
MAXTBL	MAXPTR*MAXDEF

2.1.1 BUFSIZE

"BUFSIZE" is the size of the buffer that is used to rescan the input line after each pass of macro expansion. In pseudo-English, the macro processor tests each word (a collection of 1 or more characters surrounded by "white space") to see if it is in the definition table. If it finds a match, it pushes the replacement text (the macro definition) back onto the input (represented by this buffer whose size is BUFSIZE.) Then it scans the definition as though it were the original input and checks it for other macros. So now that you see how the buffer works, you can see that as long as your macro definition does not expand to a length greater than BUFSIZE, you are fine. But in embedded builds this value my be way to generous for your limited RAM space. If that's true also consider changing MAXPTR and possibly MAXDEF, because they are used to calculate MAXTBL (the size of the definition table itself.) If your port of interp placed the macro definition table in flash (ROM) space then you can ignore changing MAXPTR, MAXDEF, and MAXTBL unless you are about to run out of flash space.

For Linux builds, the best advice, is if you get error #1 "putbak: too many characters pushed back" and you know that your definition does not contain a reference to itself (a recursive definition), then you should definitely consider making BUFSIZE larger.

2.1.2 MAXPTR

"MAXPTR" is one less than the maximum number of macro definitions that can be placed in the definition table. For Linux builds, if you are getting error #7 "*name*: too may definitions in insert_macro" consider enlarging MAXPTR. This situation usually indicates you have lots of short definitions. If that is the case, you may not need to enlarge MAXTBL (which is defined by default as a calculation) and may wish to define it as a static value (e.g. 9088.)

2.1.3 MAXDEF

"MAXDEF" represents the maximum length of one macro definition. Typically this value is at least one order of magnitude smaller than BUFSIZE to allow for nested macro definitions. However, if your macro definition does not fit in an 80-character line, you will need to increase MAXDEF. Once again, in embedded builds, you may find the default definition length is way to generous. In that case feel free to reduce it.

2.1.4 MAXTBL

"MAXTBL" is defined by default as "MAXPTR*MAXDEF". If you want the table size to be smaller but don't want to reduce, MAXPTR or MAXDEF, or if you need to increase MAXPTR but don't want a larger table, define MAXTBL as a static value (replacing the default calculation.)

2.2 Inner Interpreter Configuration Parameters (INNER_OPTIONS)
Embedded build are more likely to need these values adjusted than Linux builds because of the typically restricted RAM in embedded computer systems.

SYMBOL NAME	DEFAULT
DATA_STACK_DEPTH	1024
LCSP_STACK_DEPTH	1024
FRSP_STACK_DEPTH	1024
VARIABLE_POOL_DEPTH	1024
NO_PRIVATE_DATA	not defined
ENFORCE_ADDRESS_ALIGNMENT	defined

2.2.1 DATA_STACK_DEPTH
"DATA_STACK_DEPTH" must be large enough to accommodate all the data items that would ever need to be on the stack at the same time. Don't set it to zero. Other than that, use any value you like. The allocation unit is 4 bytes per stack item.

2.2.2 LCSP_STACK_DEPTH
"LCSP_STACK_DEPTH" represents the maximum nesting depth for loops. Don't set it to zero. Other than that, use any value you like. The allocation unit is 4 bytes per stack item.

2.2.3 FRSP_STACK_DEPTH
"FRSP_STACK_DEPTH" represents the maximum nesting depth for function calls. Don't set it to zero. Other than that, use any value you like. The allocation unit is 12 bytes per stack item.

2.2.4 VARIABLE_POOL_DEPTH
"VARIABLE_POOL_DEPTH" represents the size in 32-bit words of both the global and local variable pools (individually, not in aggregate.) This is a very generous 1024 32-bit words for each area. There is one global variable pool and at start up interp will also allocate the first local variable pool (for interactive use.) Then with each global or local function call, a new local variable pool will be created for use by the function while it runs, and then released when the function terminates. This allows you to see why it is possible to call functions as macros and avoid creating a local variable pool for each function call. It also shows you why it may be necessary to reduce VARIABLE_POOL_DEPTH to a size that fit into your reduced RAM size for your embedded device. The allocation unit is 4 bytes per item. The value must be a power of 2. Legal values are: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, and so forth.

2.2.5 NO_PRIVATE_DATA
When "NO_PRIVATE_DATE" is defined, global functions will not have their own private local variables or local functions. Also, local functions will not have their own private local variables. If you are really tight on RAM space for your embedded device, you will need to set both VARIABLE_POOL_DEPTH and NO_PRIVATE_DATA to gain complete control of the RAM used by functions.

2.2.6 ENFORCE_ADDRESS_ALIGNMENT
“ENFORCE_ADDRESS_ALIGNMENT” will prevent unaligned memory accesses when it is defined (default), but there is a slight performance penalty. If you don't want it, or if your target hardware allows unaligned accesses, then COMMENT-OUT this line. The affected operators are !, @, h!, h@, MR, MW, hR, hW, QUOTE (“), and M-QUOTE (M”).

2.3 Outer Interpreter Configuration Parameters (OUTER_OPTIONS)

SYMBOL NAME	DEFAULT
INBUF_SIZE	4097
BIGBUF_SIZE	64*1024
IIF_MAX_LENGTH	65535

2.3.1 INBUF_SIZE
"INBUF_SIZE" limits the maximum length of one line of input to interp, whether it is typed in or read from a file. This is a generous value that may never need to be adjusted for Linux builds. For embedded builds, it may be completely out of the question at its default value of 4097 bytes. Reduce it to any value you like, but be aware that this size includes the string terminator character ('\0'.)

2.3.2 BIGBUF_SIZE
"BIGBUF_SIZE" specifies the size of a static RAM buffer named bigbuf, which is used for concatenating lines of input. This is handled by the "#BUFFER" and "#EXECUTE" outer interpreter directives. Once again, this very generous size of 65,536 bytes may never need to be adjusted for Linux builds, but will very likely require reduction for embedded builds.

Linux builds can load IIF files via the “-b” command line option. The records are read individually from the file into bigbuf, and then moved into allocated memory and added to either the global or local function table as appropriate. As each global or local function record is being read from the file, the value of its length field is compared to BIGBUF_SIZE to make sure that the record will fit. If it doesn't fit, the load is aborted. If you need to load IIF files via the “-b” command line option, and you need to reduce BIGBUF_SIZE, also consider reducing IIF_MAX_LENGTH. See 2.3.2 below

2.3.3 IIF_MAX_LENGTH
“IIF_MAX_LENGTH” specifies the maximum length of the payload portion of the Global Function Record and the Local Function Record. This length includes the 1-byte function name and the function body or definition which includes the string terminator. The default maximum length is 65535 bytes which means that the function body (including the string terminator) is (by default) limited to 65534 bytes. If need to load IIF files via the “-b” command line option, be aware that the length of the IIF records interacts with the size of the static RAM buffer named bigbuf, and may prevent your IIF file from loading via the CLI. IN that unlikely event, refactor your gigantic functions to make them smaller, and recreate your IIF file. See 2.3.2 above.

3. FEATURE-LIMITED AND SPECIAL-PURPOSE VERSIONS OF INTERP

There are two build configurations that create separate executables (not named “interp” or “interp.exe”.) The first one creates a small executable called “sin” or “sin.exe”, which doesn’t have an outer-interpreter or macro processor. The second one creates a larger executable called “I2” or “I2.exe”, which has all of interp’s features plus ‘experimental” compiler capabilities.

3.1 "SIN" Parameter
There might be situations where you want an even smaller version of interp for restricted memory architectures, but you still have to have user interaction. See the “sin” rule in the Makefile, and see isin.c for more details.

3.2 "I2" Parameter
An “experimental” compiler is being created for interp. It will work on the same script files as the “interp” executable. The difference is “I2” translates the global and local function definitions into a more compact set of (new) opcodes that run faster because there is less runtime overhead.

3.3 "make" command line examples

make sin	builds a smaller, feature-limited version (“sin” or “sin.exe”)
make I2	builds the “experimental” compiler version (“I2” or “I2.exe”)

4. CLOSING REMARKS

Only define what you want to change. Live in peace.