For: dwight
Printed on: Fri, Jul 16, 1993 09:52:14
From book: manual
Document: appa
Last saved on: Thu, Jul 23, 1992 16:40:29

Appendix A: Using Assign Libraries

Although it is possible to write and create Assign flowgraphs using only the techniques mentioned so far, it can be awkward at times, especially if you wish to reuse previously written nodes and primitives. There are two tools that can make this process easier and remove the need for duplicating code every time a node is reused. These tools are `primcomp` and `primscan`. `primcomp` compiles source files created by `primscan` into libraries that can be searched during different phases of flowgraph compilation. In this way, you can archive all of the primitives in one standard area and easily reuse them without recompiling each time.

Creating Libraries with `primcomp` and `primscan`

Using primitive libraries allows you to create and archive primitives in one standard place, so you can use the primitives in many different flowgraphs without duplicating files every place the primitive is used.

Without primitive libraries you must create a `name.h` file and a `name.c` file. The `name.h` file contains the definition of the state structure *state_name*. The `name.c` file contains the definitions of the primitive function *prim_name*, the input function *in_name*, the output function *out_name*, and the dynamic initializer *init_name*.

The `primcomp` and `primscan` utilities automate the process of creating primitives for Assign flowgraph nodes. They both take as input one or more `name.prim` files, where `name` is the name of the primitive. `primcomp` and `primscan` do the following:

- `primscan` creates three files (`name.h`, `name.c`, and `name_ngf.c`) from a `name.prim` file. The files `name.h` and `name.c` contain the same functions as when they were written by hand (*state_name*, *prim_name*, *in_name*, and *out_name*), while `name_ngf.c` will contain a function called a *node-generating function*, which encapsulates the Assign function calls needed to instantiate a new node.
- `primcomp` creates three primitive libraries from the `name.prim` file. `primcomp` calls `primscan` to create the files it compiles into libraries. `primcomp` will remove the primscan `.c` files so the only files left will be the original `name.prim` and the `name.h` files, in addition to the libraries.

The `name.prim` file contains the functions and structures that were previously in the `name.h` and `name.c` files. These constructs must be delimited by special markers. The markers are listed below. Note that `$FSTATE` and `$FPRIMITIVE` are used for marking FORTRAN constructs, although in our examples we are assuming that C is used. Refer to the `primcomp` and `primscan` manual pages for a more in-depth discussion of the marker set and usage rules.

- `$STATE` marks the state structure.
  `$FSTATE` marks a FORTRAN state structure.
- `$INIT` marks the dynamic initializer.
- `$IN` marks the input function.
- `$OUT` marks the output function.
- `$PRIMITIVE` marks the primitive function.
  `$FPRIMITIVE` marks a FORTRAN primitive function.
- `$NGF` marks the node generation function.

The input to primcomp is the name.prim file. primcomp first runs primscan to create the name.h, name.c, and name_ngf.c files. Then primcomp compiles these source files to create object files and archives the object files into libraries. The libraries created are:

- libiw.a contains the object code from the name.c files, compiled for the iWarp system.
- libhost.a contains the object code from the name.c files, compiled for the host.
- libngf.a contains the object code from the name_ngf.c files (compiled for the host).

Only the libraries, name.h, and name.prim (and possibly an Index file) will be left in the directory when primcomp finishes. The names given above are the default names. Command line options allow you to change the names and locations of the libraries. Figure A-1 illustrates the primcomp process.

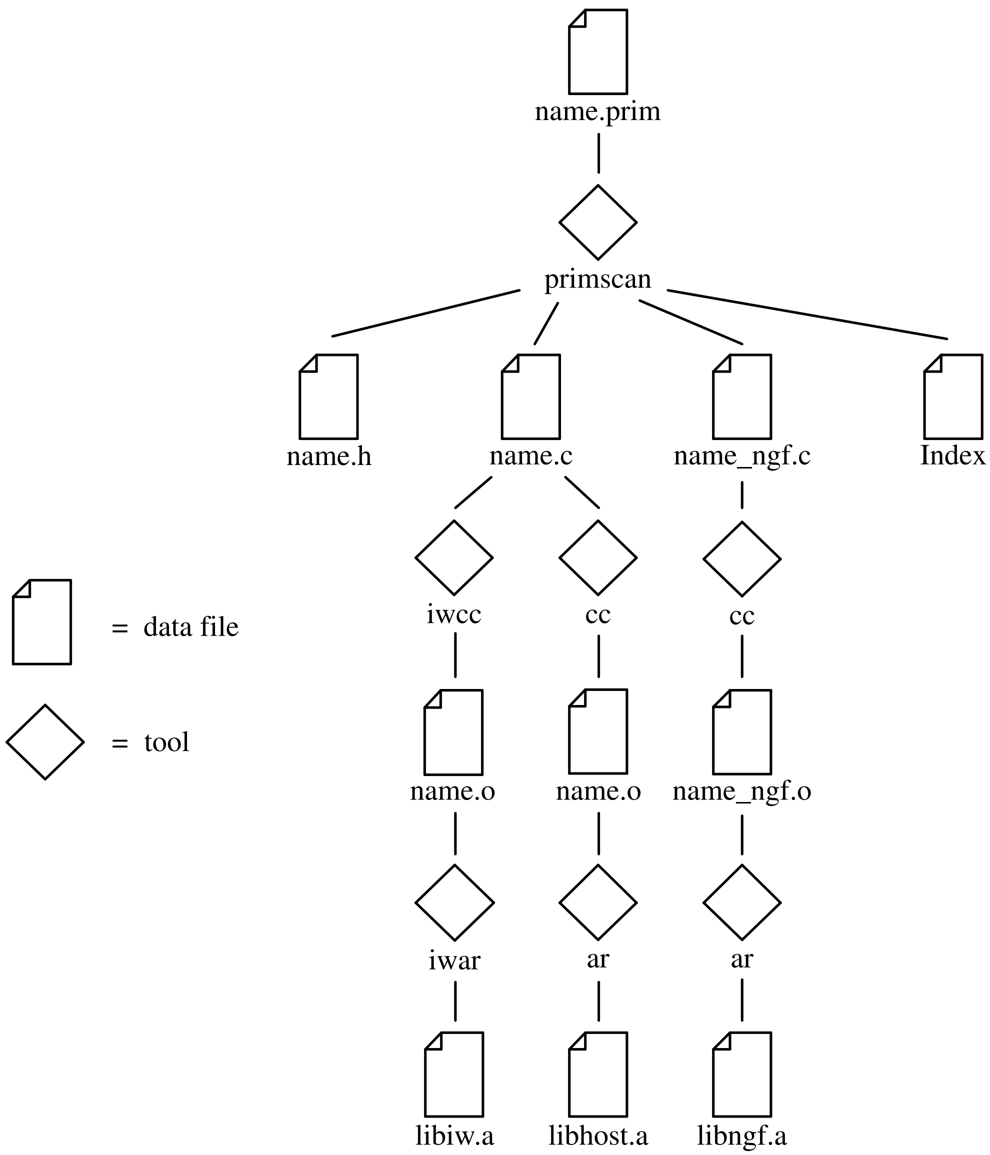

Figure A-1: The primcomp Process

Using the Libraries to Generate Flowgraphs

You can use the three libraries created by `primcomp` to generate flowgraphs. The `libiw.a` library contains routines used in the cell-side programs. That is, it contains routines that execute on the iWarp array when the flowgraph is executed. `libhost.a` also contains code for execution time, but only on the host side. These two libraries are the same, except `libiw.a` is compiled for the iWarp system, while `libhost.a` is compiled for the host. The `libngf.a` library contains the NGF functions that are used at build time, when the flowgraph is created. These only execute on the host, never on the array. You can call the NGF functions in the graph-generation program to instantiate ready-to-use nodes to use in the flowgraph being created.

Linking the Libraries

When the Assign driver is called, directives must be passed to the build and load (or host) passes of the compilation so the libraries can be linked in. `libngf.a` must be linked in during the build pass so the NGFs will be defined. This is because the graph-generating program is compiled and run in this stage to generate the `name.graph` file, which is the input to the Assign pass. You can do this by including the following option on the Assign command line:

```
-Wb,-L<library directory>,-lngf
```

`library directory` is the pathname of the directory in which `libngf.a` resides.

Similarly, because the executable flowgraph calls the functions in `libiw.a` (or `libhost.a`), this library must be linked during the load (or host) pass of the compilation so the primitive and other functions will be defined. To link `libiw.a` for the load phase, pass the following options to the Assign driver

```
-Wl,-Wl,-L<library directory>,-liw
```

To do this for the host phase, pass these options to the Assign driver:

```
-Wh,-L<library directory>,-lhost
```

`library directory` is the pathname of the directory in which the desired library resides.

NOTE

Since we are linking in libraries in the final pass of the driver, you must not use add_obj_file() to include primitive .o files at build time. Object files are not included in the array module, but are linked in during final compilation of the executables.

In sum, the use of primitive libraries allows you to maintain a large number of primitives in one place and access them easily for use in any flowgraph. The only changes in the programming steps are that the `name.h` file and the `name.c` file are combined into one `name.prim` file in which the NGF is also included, and that two more options need to be passed to the Assign driver. An example `.prim` file is detailed in the following section.

Example `name.prim` File and Compilation Steps

This section presents an example primitive file and an example build session which shows how the process changes to accommodate the use of the primitive file.

Example `cxincr.prim` File

```
CXINCR : A node that increments a vector of complex data by 1.

$STATE
/* state structure for cxincr.c */
  struct state_cxincr
{
  int size;
};

$IN
/*
 * input routine for an cxincr node
 */
#include <assign/assign.h>
in_cxincr(inp, sp)
complex *inp;
struct state_cxincr *sp;
{
  int i;

  for (i=0; i<sp->size; i++)
     {
       inp[i].r = (float)i;
       inp[i].i = (float)i;
     }
}

$OUT
/*
 * cxincr_out - output routine for a cxincr node
 */
#include <assign/assign.h>
out_cxincr(outp, sp)
complex *outp;
struct state_cxincr *sp;
{
  int i;

  for (i=0; i<sp->size; i++)
     {
      printf("(%.2f,%.2f)\n", outp[i].r, outp[i].i);
     }
}
```

$PRIMITIVE

```
/*
 * increment elements of a complex vector of size sp->size
 */
void
prim_cxincr(inp, outp, sp)
complex *inp, *outp;
struct state_cxincr *sp;
{
  int i;

  for (i=0; i < sp->size; i++)
    {
      outp[i].r = inp[i].r + 1.0;
      outp[i].i = inp[i].i + 1.0;
    }
}
```

$NGF

```
/*
 * ngf_cxincr_node() : create a node which increments
 * each element of a complex vector by one. The size
 * of the vector is n.
 */
#include <assign/assign.h>
#include <stdio.h>

NODE *ngf_cxincr_node(gp, name, n)
GRAPH *gp;
char *name;
int n;
{
  NODE *np;
  char ilist[50];

  /* Add the node to graph "gp" with name "name".
   * The primitive function which performs the run-time
   * addition is called "prim_cxincr".  The estimated
   * number of clocks for this operation is 20 * size.
   */
  np = add_node(gp, name, "prim_cxincr", n*20);

  /* There is one in and one out port. Both operate on
   * an "n" element vector of type COMPLEX. */
  add_in_port(np, "in", COMPLEX, n);
  add_out_port(np, "out", COMPLEX, n);
```

```
  /* Add a state structure and initialize the size field
   * to the size of the vector. */
  (void)sprintf(ilist, "%d", n);
  add_state(np, "state_cxincr", ilist, "");

  return(np);   /* return a ready-to-use node */
}
```

Command Line Example

This section shows how to build a flowgraph that uses the cxincr primitive shown in the previous section. In keeping with the ABI example shown in Chapter 2 and Appendix B, the name of the flowgraph will be `abi`. We will assume `abi-gen.c` is already written, and that `abi-graph.c` contains the `abi-graph()` function called in `abi-gen.c`.

We also assume that the following paths exist from the user's home directory, `~username`:

```
~username/assign/prim_lib
~username/assign/examples/abi
```

and that these files exist:

```
~username/assign/prim_lib/cxincr.prim
~username/assign/examples/abi/abi-gen.c
~username/assign/examples/abi/abi-graph.c
```

We also assume that assign, primcomp, and primscan have been installed in a standard area so pathnames for them and related files do not have to be specified. Lastly, we assume the graph is being built for execution on both the iWarp array and the host.

Making the Libraries

Given the above assumptions, we can proceed with the process of building the `abi` flowgraph. The first step is to use `primcomp` to make the primitive libraries. This should be done in the `prim_lib` directory as follows:

```
~username/assign/prim_lib> primcomp cxincr
```

The libraries created (`libiw.a`, `libhost.a`, and `libngf.a`) only contain functions for the cxincr node. They would contain other functions if other primitives were specified on the command line. If the libraries already existed, `cxincr` would be added to the existing libraries. `cxincr` is specified without the `.prim` suffix because `primcomp` automatically appends the suffix.

Building the Flowgraph

Once the libraries are created, you can proceed with the build process by compiling `abi-graph.c` into an object file, `abi-graph.o`, so the `abi-graph()` function can be linked in when `abi-gen.c` is compiled. Do this in the `abi` directory:

```
~username/assign/examples/abi> cc -c abi-graph.c
```

Then you can call Assign to build the graph:

```
~username/assign/examples/abi> assign  abi  -E,16,32 \
-Wb,-L~username/assign/prim_lib,-lngf  \
-Wc,-I~username/assign/prim_lib \
-Wl,-W1,-L~username/assign/prim_lib,-liw
```

This results in a `go` script being created which you can use to execute the flowgraph. The `-I`, `-L` and `-l` options are passed to the build, combine, and load phases because the libraries and include files (which were created when `primcomp` invoked `primscan`) are not in a standard area.

Manual Pages

For further information about `primcomp` and `primscan`, refer to the respective manual pages.