

CODE SEGMENTS

While data accesses are made via DPPs, opcode fetches are made on the basis of 64k segments. When a CALL or JUMP is made in assembler, if the call is within the current segment then the destination address is simply a 16-bit offset from the current segment base, held in the CSP (code segment pointer) register. If no segment number is given, execution speed is faster. It also means that subroutines must be grouped together in segments, unless they are called used a “segmented” call (CALLS).

Functions which are outside the current code segment are called with a CALLS instruction and must be terminated with a RETS. These functions are called “far” in C16x.

Example

Call a function at 0x18000, i.e. 0x8000 offset in code segment 0x01:

```
CALLS    01,08000H
```

Functions that are called without giving a destination segment number are called “near.”

The amount of ROM and RAM on your target, plus the balance between code and data, largely determine which model is the best choice for your application.

Definitions

Near functions	= Functions can only be called from within the same segment
Far functions	= Functions can be called across 64k segment boundaries
Near data	= Data in a single 16k range, addressed by a single DPP (DPP2) which never changes
Far data	= Data anywhere in a 256k (or 16mb) memory space; no single object may be over 16k
Huge data	= Data anywhere in the memory space; no single object may be over 64k in size
Xhuge data	= Data anywhere in the memory space; objects have unlimited size
Type qualifier	= Special keyword which when included in a data object definition can influence where it ends up in memory

C16X MEMORY MODELS

Choosing which memory model to use in your project is perhaps the most important decision to make. When considering which to use, examine carefully every aspect of your project since changing models in the middle of a project can be very difficult and time consuming.

The term “model” simply refers to how you want C16x to group together data objects and whether segmented or non-segmented function calls are used. Hopefully, the following guidelines will help:

Definitions

Default memory space = The area (near or far) where data declared without a type qualifier is stored

Class = The physical memory region occupied by data of the same type qualifier; i.e. near, far or huge

Example

```
int test_var;           /* Variable goes into near, far or huge
                        space, depending on the memory model */

int far test_var;      /* Variable goes into far data area,
                        regardless of the memory model */
```

C16x supports seven memory models (tiny, small, medium, compact, large, hlarge, and hcompact) which are described below.

TINY:

In this model, the CPU runs in segmented mode (SGTDIS = 0 in SYSCON). This means that all code and data must be within the first 64k. All function calls must be within the same segment. A16 and A17 are inactive and can be used as simple I/O pins. The DPP registers stay at their reset values.

SMALL:

In this model, the CPU runs in segmented mode so that A16 and A17 are active. All function calls are within segment “near” and all data defaults to “near” (within a 16k range). Generally, code size is limited to 64k and data is limited to 16k of variables and 16k of constants. Using the far and huge keywords in data declarations, overall data size can be expanded to any size.

MEDIUM:

In this model, the CPU runs in segmented mode so that A16 and A17 are active. All function calls are now “far” and all data defaults to “near” (within a 16k range). Code size is unlimited

and data is generally limited to 16k of variables and 16k of constants. Again, using the far and huge keywords in data declarations will give unlimited data size. This is a very useful models as it give no restriction on code sizes but data still defaults to “near” addressing (fast).

COMPACT:

This model is the reverse of MEDIUM as data defaults to “far” and functions default to “near”. This is useful for programs with small amounts of code but a large data group.

LARGE:

This model treats both function calls and data objects as “far”. It suits large applications and is perhaps the safest choice if the final program characteristics are not obvious.

HLARGE:

This model is only available with MOD167. HLARGE is the same as LARGE but the default for variable declarations is “huge.” “Huge” addressing is handled more efficiently than “far” and so this model is often preferred to LARGE.

HCOMPACT:

This model is only available with MOD167. HCOMPACT is the same as COMPACT but, again the variable declarations default to “huge” and addressing is handled more efficiently than “far.” This models is also often preferred to COMPACT.

Note: For C167 applications with large amounts of data, the HLARGE model combines the HOLD control to force small objects (char, int, short, long) into the NEAR area, effectively giving the easiest development route. This is particularly true of programs ported from 16 bit PC compilers as the characteristics of the default huge point type is identical.

TYPE QUALIFIERS

The chosen memory model determines the location of data objects. In many cases, though, it is useful to override this default placement. An example in a SMALL model program might be to put a large array into the far data area so that all the fast near data area is not used up. The “far” type qualifier is used to achieve this.

The syntax for using type qualifiers is:

```
<type> <type qualifier> <object name> = <initialization value>
```

Example

```
int far test_var = 0;
```

You may use the type qualifiers near, far, huge, xhuge, idata, bdata, and sdata to manually place variables into specific memory areas. Each of these qualifiers is described below.

near:

Purpose: Allow data to be forced into reasonably fast access area.

Overall size of all near data: 16k

Largest single object: 16k

CLASS name: NDATA, NDATA0

far:

Purpose: Allow a large number of small arrays to be grouped together. This keyword is best replaced by “huge” on the C167.

Overall size of all far data: 16m

Largest single object: 16k

CLASS name: FDATA, FDATA0

huge:

Purpose: Allow use of large data objects.

Overall size of all huge data: 16m

Largest single object: 64k

CLASS name: HDATA, HDATA0

xhuge:

Purpose: Allow very large data objects to be used.

Overall size of all xhuge data: 16m

Largest single object: 16m

CLASS name: XDATA, XDATA0

idata:

Purpose: Force data into on-chip RAM. This RAM is always addressed at full speed, regardless of the external bus type or wait-states etc.

Overall size of all idata objects: determined by CPU type

Largest single object: 16k (not realizable)

CLASS name: IDATA, IDATA0

bdata:

Purpose: Combined word and bit-addressable data. Can be used in conjunction with the sbit control to allow bit addressing of individual bits in an integer (byte).

Overall size of all bdata objects: 256b

Largest single object: 256b

CLASS name: BDATA, BDATA0

sdata:

Purpose: Force data into the area between 0xC000 and 0xFFFF, to be addressed via DPP3 which is always set to page 3. Useful for addressing PEC pointers and memory-mapped I/O at 0xC000. On the C167CD/SR, sdata objects can be used to fill the XRAM area.

Overall size of all sdata objects: approx. 0x3000

Largest single object: approx. 0x3000

CLASS name: SDATA, SDATA0

Copyright © 1997-1998 Keil Software, Inc. All rights reserved.

In the USA:
Keil Software, Inc.
16990 Dallas Parkway, Suite 120
Dallas, TX 75248-1903
USA

Sales: 800-348-8051
Phone: 972-735-8052
FAX: 972-735-8055

E-mail: sales.us@keil.com
support.us@keil.com

Internet: <http://www.keil.com/>

In Europe:
Keil Elektronik GmbH
Bretonischer Ring 15
D-85630 Grasbrunn b. Munchen
Germany

Phone: (49) (089) 45 60 40 - 0
FAX: (49) (089) 46 81 62

E-mail: sales.intl@keil.com
support.intl@keil.com